MaDcAr-Agent : un modèle d'agents auto-adaptables à base de composants

Guillaume Grondin

Ecole Nationale Supérieure des Mines
SAINT-ETIENNE

N° d'ordre : 499 I

Thèse

présentée par

Guillaume Grondin

pour obtenir le grade de
Docteur de l'École Nationale Supérieure des Mines de Saint-Étienne

Spécialité : Informatique

MaDcAr-Agent : un modèle d'agents auto-adaptables à base de composants

Soutenue à Douai, le 24 novembre 2008

Membres du jury

Président :
- Stéphane Ducasse — Directeur de recherches, INRIA, équipe RMOD , Lille

Rapporteurs :
- Jean-Pierre Briot — Directeur de recherches, CNRS, LIP6, Paris
- Michel Occello — Professeur, Univ. Pierre Mendès France, LCIS, Valence

Examinateurs :
- Jean-Paul Arcangeli — Maître de conférences, Univ. Paul Sabatier, IRIT, Toulouse
- Sylvain Lecomte — Professeur, Univ. de Valenciennes, LAMIH, Valenciennes

Directeurs de thèse :
- Olivier Boissier — Professeur, ÉNS des Mines, Saint-Étienne
- Noury Bouraqadi — Maître assistant, ÉNS des Mines, Douai
- Laurent Vercouter — Maître assistant, ÉNS des Mines, Saint-Étienne

Spécialités doctorales :

SCIENCES ET GÉNIE DES MATÉRIAUX
MÉCANIQUE ET INGÉNIERIE
GÉNIE DES PROCÉDÉS
SCIENCES DE LA TERRE
SCIENCES ET GÉNIE DE L'ENVIRONNEMENT
MATHÉMATIQUES APPLIQUÉES
INFORMATIQUE
IMAGE, VISION, SIGNAL
GÉNIE INDUSTRIEL
MICROÉLECTRONIQUE

Responsables :

J. DRIVER Directeur de recherche - Centre SMS
A. VAUTRIN Professeur - Centre SMS
G. THOMAS Professeur - Centre SPIN
B. GUY Maître de recherche - Centre SPIN
J. BOURGOIS Professeur - Centre SITE
É. TOUBOUL Ingénieur - Centre G2I
O. BOISSIER Professeur - Centre G2I
JC. PINOLI Professeur - Centre CIS
P. BURLAT Professeur - Centre G2I
Ph. COLLOT Professeur - Centre CMP

Enseignants-chercheurs et chercheurs autorisés à diriger des thèses de doctorat (titulaires d'un doctorat d'État ou d'une HDR)

AVRIL	Stéphane	MA	Mécanique & Ingénierie	CIS
BATTON-HUBERT	Mireille	MA	Sciences & Génie de l'Environnement	SITE
BENABEN	Patrick	PR 2	Sciences & Génie des Matériaux	CMP
BERNACHE-ASSOLANT	Didier	PR 1	Génie des Procédés	CIS
BIGOT	Jean-Pierre	MR	Génie des Procédés	SPIN
BILAL	Essaïd	MR	Sciences de la Terre	SPIN
BOISSIER	Olivier	PR 2	Informatique	G2I
BOUCHER	Xavier	MA	Génie Industriel	G2I
BOUDAREL	Marie-Reine	MA	Sciences de l'inform. & com.	DF
BOURGOIS	Jacques	PR 1	Sciences & Génie de l'Environnement	SITE
BRODHAG	Christian	MR	Sciences & Génie de l'Environnement	SITE
BURLAT	Patrick	PR 2	Génie industriel	G2I
CARRARO	Laurent	PR 1	Mathématiques Appliquées	G2I
COLLOT	Philippe	PR 1	Microélectronique	CMP
COURNIL	Michel	PR 1	Génie des Procédés	SPIN
DAUZERE-PERES	Stéphane	PR 1	Génie industriel	CMP
DARRIEULAT	Michel	ICM	Sciences & Génie des Matériaux	SMS
DECHOMETS	Roland	PR 1	Sciences & Génie de l'Environnement	SITE
DESRAYAUD	Christophe	MA	Mécanique & Ingénierie	SMS
DELAFOSSE	David	PR 1	Sciences & Génie des Matériaux	SMS
DOLGUI	Alexandre	PR 1	Génie Industriel	G2I
DRAPIER	Sylvain	PR 2	Mécanique & Ingénierie	CIS
DRIVER	Julian	DR	Sciences & Génie des Matériaux	SMS
FOREST	Bernard	PR 1	Sciences & Génie des Matériaux	CIS
FORMISYN	Pascal	PR 1	Sciences & Génie de l'Environnement	SITE
FORTUNIER	Roland	PR 1	Sciences & Génie des Matériaux	CMP
FRACZKIEWICZ	Anna	MR	Sciences & Génie des Matériaux	SMS
GARCIA	Daniel	CR	Génie des Procédés	SPIN
GIRARDOT	Jean-Jacques	MR	Informatique	G2I
GOEURIOT	Dominique	MR	Sciences & Génie des Matériaux	SMS
GOEURIOT	Patrice	MR	Sciences & Génie des Matériaux	SMS
GRAILLOT	Didier	DR	Sciences & Génie de l'Environnement	SITE
GROSSEAU	Philippe	MR	Génie des Procédés	SPIN
GRUY	Frédéric	MR	Génie des Procédés	SPIN
GUILHOT	Bernard	DR	Génie des Procédés	CIS
GUY	Bernard	MR	Sciences de la Terre	SPIN
GUYONNET	René	DR	Génie des Procédés	SPIN
HERRI	Jean-Michel	PR 2	Génie des Procédés	SPIN
KLÖCKER	Helmut	MR	Sciences & Génie des Matériaux	SMS
LAFOREST	Valérie	CR	Sciences & Génie de l'Environnement	SITE
LI	Jean-Michel	EC (CCI MP)	Microélectronique	CMP
LONDICHE	Henry	MR	Sciences & Génie de l'Environnement	SITE
MOLIMARD	Jérôme	MA	Sciences & Génie des Matériaux	SMS
MONTHEILLET	Frank	DR 1 CNRS	Sciences & Génie des Matériaux	SMS
PERIER-CAMBY	Laurent	PR 1	Génie des Procédés	SPIN
PIJOLAT	Christophe	PR 1	Génie des Procédés	SPIN
PIJOLAT	Michèle	PR 1	Génie des Procédés	SPIN
PINOLI	Jean-Charles	PR 1	Image, Vision, Signal	CIS
STOLARZ	Jacques	CR	Sciences & Génie des Matériaux	SMS
SZAFNICKI	Konrad	CR	Sciences de la Terre	SITE
THOMAS	Gérard	PR 1	Génie des Procédés	SPIN
TRAN MINH	Cahn	MR	Génie des Procédés	SPIN
VALDIVIESO	Françoise	CR	Génie des Procédés	SPIN
VALDIVIESO	François	MA	Sciences & Génie des Matériaux	SMS
VAUTRIN	Alain	PR 1	Mécanique & Ingénierie	SMS
VIRICELLE	Jean-Paul	MR	Génie des procédés	SPIN
WOLSKI	Krzysztof	CR	Sciences & Génie des Matériaux	SMS
XIE	Xiaolan	PR 1	Génie industriel	CIS

Glossaire :

PR 1	Professeur 1ère catégorie
PR 2	Professeur 2ème catégorie
MA(MDC)	Maître assistant
DR(DR1)	Directeur de recherche
Ing.	Ingénieur
MR(DR2)	Maître de recherche
CR	Chargé de recherche
EC	Enseignant-chercheur
ICM	Ingénieur en chef des mines

Centres :

SMS	Sciences des Matériaux et des Structures
SPIN	Sciences des Processus Industriels et Naturels
SITE	Sciences Information et Technologies pour l'Environnement
G2I	Génie Industriel et Informatique
CMP	Centre de Microélectronique de Provence
CIS	Centre Ingénierie et Santé

Remerciements

Je tiens à remercier ici tout un ensemble de personnes grâce à qui, directement ou indirectement, j'ai pu mener ce travail à bien.

Je remercie tout d'abord Jean-Pierre Briot et Michel Occello qui m'ont fait l'honneur d'être les rapporteurs de cette thèse. Je remercie également Jean-Paul Arcangeli, Stéphane Ducasse et Sylvain Lecomte d'avoir bien voulu participer à ce jury.

Je remercie Olivier Boissier pour son rôle de directeur de thèse. Il a su poser les bonnes questions afin de recadrer mon travail lors des moments importants. Je remercie très chaleureusement Noury Bouraqadi et Laurent Vercouter de m'avoir accompagné et guidé durant ces quatre années. Je leur suis infiniment reconnaissant pour le temps qu'ils m'ont accordé et pour les conseils qu'ils m'ont prodigués.

Je tiens à remercier Philippe Hasbroucq de m'avoir accueilli dans le département Informatique et Automatique (IA) de l'EMD. La convivialité qui règne dans ce département a facilité mon intégration aussi bien dans l'équipe recherche que dans l'équipe pédagogique. Je remercie également son adjoint chargé de la recherche, Stéphane Lecœuche, qui m'a régulièrement encouragé au cours de ma dernière année de thèse. À travers eux, je remercie tout le personnel que j'ai côtoyé pendant ma thèse (enseignants-chercheurs, techniciens, secrétaires, ...).

Je salue particulièrement les gens que j'ai rencontrés au département IA pour le travail et qui resteront pour moi des amis : Arnaud, Djamel, Houssam, Aziz, Gautier (et ses parents), Sylvain (et sa famille), Tuan, Nikolay, Ludovic, Mickaël, Rémy, Misagh, « les automaticiens » et tous ceux que j'oublie.

Je salue également les membres de l'équipe SMA que j'ai pu rencontrer pendant mon séjour à l'EMSE : Rahee, Guillaume, Maxime, ...

Je remercie l'ADMID, notamment son actuel président Tudor Floréa, d'avoir organisé des actions à l'intention des doctorants. Merci aussi aux membres (souvent les mêmes) qui veulent bien prendre part aux activités collectives, en particulier les soirées cinéma, bowling et poker (« C'est la main de .... ! »).

Je remercie également la Direction de la Recherche à l'EMD d'avoir apporté son soutien aux différentes actions de l'ADMID.

Je n'oublie pas de penser aux enseignants que j'ai eus par le passé, depuis l'école primaire à la Réunion jusqu'au DEA à Nantes. En particulier, je remercie tous ceux qui m'ont encouragé à faire de la recherche.

Je remercie Rémy Pinot, chef de projet à l'EMD, qui m'a proposé de jouer dans le Volley-Ball Club de Cuincy. Je remercie le président Alain Hornez et tous les membres de m'avoir fait une place parmi eux. Les périodes de compétition et les nombreux instants de convivialité ont rythmé agréablement ma vie extra-scolaire.

Enfin, je remercie mon père, ma mère, mes frères et mes sœurs pour leur aide morale et financière. Merci en particulier pour les cartes postales « couleur volcan » et les colipays ;-)

Mes chers parents, votre courage a nourri ma persévérance. C'est pourquoi, je vous dédie cette thèse.

Résumé

Dans le cadre de l'informatique ubiquiste, l'environnement d'exécution d'une application est constitué de machines hétérogènes en ressources matérielles et appartenant à des utilisateurs différents (PC, PDA, téléphone mobile, etc.). Ces caractéristiques imposent de structurer l'application en une organisation d'unités logicielles relativement indépendantes qui coopèrent et interagissent. Dans cette thèse, nous proposons MaDcAr-Agent, un modèle d'agents auto-adaptables à base de composants et muni d'une infrastructure dédiée à l'adaptation. Ce modèle se caractérise par la présence d'un niveau méta qui comporte notamment un moteur d'assemblage en charge des adaptations dynamiques et automatiques en fonction du contexte de l'agent. Le fonctionnement du niveau méta est guidé par la spécification de deux politiques : la *politique d'assemblage* qui permet à l'agent de s'adapter aux changements de contexte en fonction des composants disponibles et la *politique de gestion de contenu* qui permet à l'agent d'avoir les composants dont il a le plus besoin grâce aux interactions avec les autres agents. A travers ces spécifications explicites et découplées du comportement applicatif de l'agent, le concepteur d'agents peut prendre en charge la perturbation d'un système dû à des changements imprévus et répétés, sans pour autant nuire à l'autonomie des agents qui composent ce système. Pour valider notre approche, diverses expérimentations ont été menées avec ce modèle, notamment dans le cadre d'un scénario impliquant des robots mobiles qui doivent explorer une zone inconnue.

Mot-clefs : Système Multi-Agent, Composants logiciels, Auto-adaptation, Adaptation dynamique, Système ouvert, Moteur d'assemblage.

Abstract

In the context of ubiquitous computing, the application execution environment is made of heterogeneous devices belonging to different users (PC, PDA, mobile phone, etc.). These characteristics require to structure the application in a network of software units that are relatively independent but may interact with each other. In this thesis, we propose MaDcAr-Agent, a model of self-adaptive component-based agents provided with a framework dedicated to adaptation. This model is characterized by a meta level which contains an assembling engine in charge of automatic and dynamic adaptations according to the agent's context. The behavior of the meta level is ruled by two specified policies: the *assembling policy* that allows an agent to adapt itself because of contextual changes according to the available components and the *content management policy* that allows an agent to obtain the components it mostly needs, by using interactions with other agents. These specifications are made explicit and uncoupled from the agent's applicative behavior. The agent designer uses them to take into account the disturbance of the system from unpredictible and repeated changes, without having a detrimental effect on the agent's autonomy. To validate our approach, several experimentations have been conducted with this model. For instance, we describe a scenario involving mobile robots that must explore autonomously an unknown area.

Keywords : Multi-Agent System, Software Components, Self-adaptation, Dynamic Adaptation, Open System, Assembling Engine .

Table des matières

Table des figures

Liste des tableaux

Introduction

Chapitre 1

Introduction

De nos jours, la miniaturisation aidant, la plupart des dispositifs qui nous entourent sont équipés de microprocesseurs. Cela va de la voiture à la machine à laver en passant par le téléphone et autre assistant numérique. Par ailleurs, grâce à l'avènement des réseaux sans-fil, ces dispositifs sont de plus en plus dotés d'équipements de communication. Dès lors, ces différents équipements sont en mesure de dialoguer et d'interagir entre eux et avec le monde environnant. Cette tendance semble devoir s'accentuer dans le futur, notamment avec les nanotechnologies pour déboucher sur un monde où l'informatique serait omniprésente et invisible, c'est l'*informatique ubiquiste* (ou ubiquitaire) [Wei93].

Dans cet univers, l'environnement des machines serait sans cesse changeant. En effet, comme c'est le cas actuellement avec les téléphones mobiles, une partie des équipements suivrait leurs propriétaires dans leurs déplacements. Les logiciels de ces machines doivent ainsi s'adapter automatiquement à ces changements de contexte (*i.e.* sans intervention de l'utilisateur) et à chaud (*i.e.* sans arrêter l'exécution).

Une autre facette de l'informatique ubiquiste est relative aux ressources des machines. La plupart des petits dispositifs disposent de peu de ressources. C'est le cas aussi bien en terme de puissance de traitement (CPU), de mémoire, d'autonomie énergétique ou encore de connectivité à un réseau. Cette caractéristique fondamentale doit être prise en compte dans les infrastructures logicielles de l'informatique ubiquiste.

Le développement des applications informatiques s'inscrivant dans ce contexte mobilise différentes technologies. Parmi celles-ci, nous pouvons notamment citer la technologie des composants logiciels et celles des systèmes multi-agents. Ces deux technologies définissent le cadre de nos travaux.

La technologie des *composants logiciels* [Szy98] est née d'une demande

de plus en plus forte de réutilisation du code lors du développement d'applications informatiques. L'idée principale est de s'inspirer de l'électronique ou de la mécanique pour lesquelles il existe un catalogue de composants industriels que l'on assemble ou adapte en fonction des besoins du produit à réaliser. Ainsi, la technologie des composants veut promouvoir le développement d'applications par assemblage de composants logiciels « disponibles sur étagères ».

Le domaine des *Systèmes Multi-Agents* (SMAs) [JW98] s'intéresse à la conception de réseaux d'entités autonomes et coopérantes. Les entités logicielles actives, appelées agents, bénéficient d'une grande autonomie de décision dans leurs interactions avec les autres agents, dans leurs actions sur (ou leur interprétation de) l'environnement, dans la place qu'ils occupent dans l'organisation du réseau, ainsi que dans leurs comportements vis-à-vis d'un utilisateur. Cette vision décentralisée du système global favorise sa faculté d'évolutivité et d'adaptabilité, tout en facilitant la prise en compte de réseaux de moyenne à grande échelle.

Composants logiciels et systèmes multi-agents ont en commun l'intérêt de permettre la structuration d'un système logiciel en une organisation d'unités logicielles relativement indépendantes et qui interagissent. L'utilité de ces deux technologies est particulièrement notable pour l'informatique ubiquiste à cause des besoins en autonomie et en adaptation. Le besoin en autonomie se trouve notamment dans le fait que les différents dispositifs matériels impliqués dans une application doivent interagir en minimisant à la fois les interventions humaines (par exemple, récupérer automatiquement des informations en fonction des préférences d'un utilisateur) et le couplage entre ces dispositifs (car chaque dispositif peut s'arrêter à tout moment, volontairement ou inopinément). Le besoin en adaptation résulte du fait que les machines impliquées ont des ressources matérielles hétérogènes et que leurs contextes d'exécution évoluent sans cesse.

Problématique

Dans le contexte décrit ci-dessus, nous sommes confrontés à des problèmes qui nous poussent à utiliser conjointement la programmation par composants et la conception d'agents. Nous souhaitons pouvoir adapter dynamiquement et automatiquement les agents à base de composants, en particulier, grâce à des opérations de restructuration (ou reconfiguration) : l'ajout, la suppression et le remplacement dynamique de composants. Cette problématique implique différents verrous technologiques tant au niveau de la programmation par composants que de la conception d'agents.

Besoins pour l'adaptation au niveau de la programmation par composants

Les opérations d'adaptation ne sont pas faciles à mettre en œuvre dynamiquement, car elles peuvent gêner l'exécution de l'application, voire introduire des incohérences dans l'application. Dans beaucoup d'applications ou de systèmes industriels, la mise en œuvre des reconfigurations est prise en charge directement dans l'implémentation de l'application. Ainsi, du code pour les adaptations prévues doit être mélangé au code applicatif. Ce mélange des préoccupations a plusieurs inconvénients :

- la tâche des développeurs est beaucoup plus compliquée (car le code perd en clarté et en efficacité à cause des mécanismes d'adaptation employés de manière *ad hoc* pour prendre en compte des préoccupations non prévues initialement) ;
- certains composants deviennent trop spécifiques à l'application pour être réutilisés ailleurs ;
- l'architecture de l'application finit par diverger de plus en plus des spécifications initiales du concepteur.

Différentes infrastructures pour la construction et l'adaptation d'applications à base de composants réduisent ces inconvénients [MDK94, MT97, PBJ98, Sen03, Dav05, MG05]. Cependant, nous pouvons faire état de plusieurs freins au principe de réutilisabilité, qui est l'objectif majeur de la programmation par composants :

- ces travaux sont spécifiques à un modèle de composants : en grande majorité, il s'agit de modèles de composants peu répandus ;
- la spécification d'une application est non réutilisable avec un autre jeu de composants que celui prévu initialement, car utilisant des références directes vers les composants à assembler [1] ;
- les spécifications des adaptations sont souvent fortement couplées à une application particulière. Il en résulte un manque d'abstraction qui oblige l'administrateur de l'application à mettre plus l'accent sur les aspects fonctionnels de l'adaptation (*i.e.* avoir l'assemblage approprié à la fin de l'adaptation) que sur les aspects extra-fonctionnels de l'adaptation (comme la performance de l'adaptation et la maîtrise de la consommation de ressources matérielles par l'infrastructure d'adaptation).

1. sauf certains travaux utilisant un mécanisme de connexion flexible tel que des connecteurs.

Besoins au niveau de la conception d'agents auto-adaptables

Dans la plupart des modèles d'agents, l'adaptation des agents n'est pas traitée spécifiquement. Il est vrai que les agents se comportent généralement en fonction du contexte qu'ils perçoivent, mais l'espace défini par les comportements possibles est généralement fermé. C'est pourquoi, lorsque les changements de contexte apparaissent de manière non anticipée, on n'a aucune assurance que l'agent puisse s'accommoder de toutes les situations sans changer sensiblement sa structure et son comportement. Certaines architectures d'agents - à couches notamment [RW91, Fer92a, Mül96] - ont été proposées avec des mécanismes dédiés à l'auto-adaptation. Néanmoins, les adaptations prises en charge ne concernent que le comportement de l'agent. La rigidité de la structure des agents restreint leur domaine d'utilisation et rend difficile leur réutilisation dans un cadre sensiblement différent.

Récemment, plusieurs modèles d'agents auto-adaptables se sont appuyés sur une architecture plus flexible, en particulier des modèles d'agents à base de composants [Ver04, Sec03, BM05, LA06, BMP06]. Nous nous inscrivons dans cette mouvance pour pouvoir adapter aussi bien le comportement de l'agent que son architecture. De plus, nous voulons permettre au concepteur d'un agent de spécifier à travers des concepts de haut-niveau (rôle, configuration, politique) quand et comment l'agent doit s'adapter. Ces éléments doivent promouvoir le découplage entre les spécifications du concepteur et les composants à utiliser (abstraction vis-à-vis des composants). Enfin, nous voulons séparer les spécifications des fonctionnalités (configurations) et les spécifications des adaptations (politiques) pour encourager leur réutilisation dans des cadres applicatifs différents.

Thèse défendue

Dans ce travail, nous défendons la thèse que pour aborder le problème de l'adaptation d'applications ubiquitaires, il est nécessaire de fournir une infrastructure à composants pour simplifier *la construction d'agents par assemblage de composants* et *l'adaptation de ces agents de manière autonome*. En plus de la préservation de l'autonomie des agents lors des adaptations, nous abordons différents problèmes :

- *abstraction* : se détacher des spécificités de chaque modèle de composants ;
- *dynamicité* : réaliser les adaptations dynamiquement (*i.e.* sans stopper complètement l'exécution de l'agent) ;
- *ouverture* : pouvoir configurer le processus d'adaptation et permettre

aussi bien les adaptations légères (comme une modification partielle du comportement) que les adaptations profondes (comme une modification totale de la structure) ;
- *réutilisation* : favoriser le principe de réutilisation en limitant le couplage entre la spécification de l'agent, la spécification des adaptations et les composants utilisés ;
- *sensibilité au contexte* : pour déclencher et décider des adaptations en fonction de l'évolution du contexte (CPU, mémoire, connectivité, etc.).

Notre proposition s'articule autour de deux axes : la programmation par composants et les systèmes multi-agents.

En ce qui concerne la programmation par composants, notre objectif est de fournir une solution générale au problème de l'assemblage automatique et dynamique de composants. Nous proposons ainsi MaDcAr, un modèle de moteur d'assemblage de composants. Ce moteur permet la conception d'applications auto-adaptables, le moteur étant considéré comme partie intégrante de l'application.

En ce qui concerne le deuxième axe relatif aux systèmes multi-agents, nous proposons MaDcAr-Agent, un modèle d'agents auto-adaptables. Comme son nom l'indique, il s'agit d'intégrer dans chaque agent le moteur MaDcAr en charge des adaptations. Cependant le concept d'agent étant plus riche qu'une application classique, nous avons pris en compte les spécificités inhérentes au domaine des SMAs notamment en ce qui concerne l'interaction et la coopération entre agents. Ainsi, notre proposition consiste également en un protocole d'échange de composants entre agents afin de permettre à un agent d'obtenir des composants supplémentaires de la part d'autres agents, en fonction de ses besoins. Les échanges sont configurables à travers une politique de gestion du contenu de l'agent.

Plan du mémoire

Ce document comporte trois parties : la première constitue un état de l'art des problématiques que nous étudions, la deuxième décrit nos différentes propositions et la troisième valide nos propositions à travers différents scenarii.

État de l'art Notre état de l'art est composé de deux chapitres qui préfigurent les deux principaux axes de notre contribution. Le premier étudie l'adaptabilité dans le cadre des applications à base de composants tandis que

le second étudie l'adaptabilité dans le cadre des agents. Chaque chapitre finit sur une grille comparative des travaux les plus représentatifs du domaine.

- Le chapitre 2 dresse un état de l'art de l'adaptabilité des applications à base de composants logiciels. Après un rappel des concepts clés et des objectifs de la programmation par composants, nous décrivons les différentes dimensions de la problématique d'adaptation logicielle. Puis, nous reprenons ces différentes dimensions et nous les appliquons à l'étude de l'adaptation dans le cas des applications à base de composants. Ensuite, nous mettons en évidence plusieurs axes pour le perfectionnement des adaptations d'applications à base de composants. Ces différents axes sont utilisés pour décrire et comparer les travaux les plus significatifs sur l'adaptation dynamique des applications à base de composants, notamment les travaux liés à des langages de description d'architecture et ceux liés à des modèles de composants. Le chapitre se termine sur une comparaison des différents travaux présentés.
- Le chapitre 3 dresse un état de l'art de l'adaptabilité des agents. Nous commençons par expliciter la notion d'architecture d'agents. Le fait de spécifier une architecture explicite pour un agent facilite sa conception et son développement en mettant l'accent sur certaines préoccupations (raisonnement, interaction, mobilité, etc.). Nous voulons étudier comment les modèles existants permettent aux agents d'adapter non seulement leur comportement mais aussi leur architecture. C'est pourquoi, nous nous intéressons particulièrement aux modèles d'agents à base de composants. Le chapitre se termine sur une comparaison des différents travaux présentés, en distinguant les modèles d'agents auto-adaptables qui utilisent ou non la notion de composant logiciel.

Contribution La partie consacrée à nos contributions contient un chapitre dédié au premier axe de nos contributions, la programmation par composants, et un chapitre dédié au deuxième axe relatif aux systèmes multi-agents. Chacun de ces axes est comparé aux travaux présentés dans l'état de l'art.

- Le chapitre 4 décrit une nouvelle infrastructure d'adaptation pour les applications à base de composants. D'abord, nous introduisons le concept de moteur d'assemblage et les critères caractérisant ce genre d'infrastructure d'adaptation. Ensuite, nous proposons MADCAR[2], le modèle de moteurs d'assemblage. Les différentes parties du modèle sont détaillées (moteur d'assemblage, spécification des assemblages valides, gestion du contexte, politique d'assemblage, processus d'assemblage, gestion de la dynamicité). Dans ce modèle, nous traitons particulière-

2. « Model for Automatic and Dynamic Component Assembly Reconfiguration ».

ment de la gestion de l'état d'une application lors des réassemblages. Le chapitre se termine après avoir comparé notre approche à l'état de l'art et synthétisé les intérêts de notre approche.

– Le chapitre 5 décrit MaDcAr-Agent, le modèle pour les agents auto-adaptables à base de composants. Ce modèle est indépendant de tout modèle spécifique de composants logiciels car il repose sur le modèle MaDcAr pour les reconfigurations. MaDcAr-Agent est un enrichissement de MaDcAr qui étend les possibilités de l'adaptation en utilisant les capacités propres à un agent logiciel évoluant dans un système multi-agent (notamment la coopération). La première partie de ce chapitre donne une vue globale de ce modèle d'agents en décrivant les différents niveaux de son architecture. Puis, nous décrivons les mécanismes qui contrôlent la capacité d'un agent à gérer de manière autonome les composants qu'il contient (compétences et connaissances) en fonction de leur utilité, notamment en s'appuyant sur les échanges avec les autres agents. Puis, nous illustrons ce qui caractérise le plus notre modèle d'agents en explicitant des petits exemples. Ensuite, nous proposons une évaluation de notre modèle et nous le comparons à l'état de l'art. Enfin, nous fournissons un résumé de notre approche en récapitulant les principales caractéristiques du modèle.

Réalisation et validation Cette partie contient un unique chapitre.

– Le chapitre 6 qui décrit la mise en œuvre de nos contributions commence par expliciter l'implémentation du framework utilisé pour les modèles MaDcAr et MaDcAr-Agent. Puis, nous présentons un premier cas applicatif (une horloge adaptable) qui a été implémenté pour étudier la faisabilité de notre approche (rôles, configurations et résolution à base de contraintes dans une version simplifiée). Dans un deuxième cas applicatif (un client de discussion adaptatif), nous illustrons les autres aspects du modèle MaDcAr (déclenchement automatique, politique d'assemblage et transfert d'état). Enfin, pour valider le modèle MaDcAr-Agent, nous présentons un dernier cas applicatif qui décrit un scénario d'agents auto-adaptables représentant des robots mobiles.

A la fin du mémoire, nous faisons le bilan de notre approche et nous évoquons quelques possibles perspectives.

Première partie

Etat de l'art

Chapitre 2

Adaptabilité des applications à base de composants logiciels

La **programmation par composants** est un paradigme qui désigne « le développement de systèmes en tant qu'assemblages de parties (composants), le développement de parties en tant qu'entités réutilisables ainsi que la maintenance et l'amélioration de systèmes en personnalisant et en remplaçant ces parties » [Crn01]. La programmation par composants est destinée à faire face à la complexité de plus en plus manifeste des applications actuelles.

Plusieurs objectifs sont visés par ce paradigme. Tout d'abord, la programmation par composants permet d'*améliorer la modélisation d'une application*. En effet, toutes les applications non triviales comportent différentes parties qui interagissent. La modélisation d'une application par un assemblage de composants permet de représenter clairement les grandes parties qui la composent (avec des composants) et leurs interactions au travers de la structure de cet assemblage. De plus, la programmation par composants permet d'*améliorer la réutilisation du code produit*, puisqu'un composant est conçu pour fonctionner dans plusieurs contextes applicatifs. Enfin, la programmation par composants permet de *tenir compte de l'ensemble des étapes du cycle de vie des applications* tel qu'on le modélise en génie logiciel (de la conception jusqu'à l'exécution). Par conséquent, le principe de la programmation par composants est de pouvoir intégrer de manière simple des entités logicielles provenant de différentes sources, c'est-à-dire aller plus loin (et plus vite) dans la réutilisation.

Dans le cycle de vie d'un logiciel (pendant ou après la phase de développement), il peut arriver que les spécifications initiales ne soient plus pertinentes. C'est ainsi que s'impose le besoin d'adaptation des logiciels, en particulier des applications à base de composants. L'adaptation d'une application peut se révéler coûteuse en termes de maintenance et de production du logiciel.

Pour minimiser ces coûts, il convient d'intégrer la problématique d'adaptation dans le cycle de vie du logiciel, en particulier pendant l'étape d'exécution. Ritzau et Andersson ajoutent que « le plus important est de concevoir des systèmes dont les améliorations de maintenance préservent la clarté logique du système, ne serait-ce que pour étendre considérablement la vie du système » [RA00].

Dans ce chapitre, nous comparons les différentes approches existantes pour l'adaptation d'application à base de composants. Dans la section 2.1, nous faisons un rappel des concepts clés et des objectifs de la programmation par composants. Dans la section 2.2, nous introduisons la problématique de l'adaptation logicielle et nous identifions les différentes dimensions d'étude du concept d'adaptabilité. Ensuite, nous reprenons ces différentes dimensions et nous les appliquons à l'étude de l'adaptation dans le cas des applications à base de composants : il s'agit de la section 2.3. À la fin de la section 2.3, nous mettons en évidence plusieurs axes pour le perfectionnement des adaptations d'applications à base de composants. Ces différents axes sont utilisés dans la section 2.4 pour décrire et comparer les travaux les plus significatifs sur l'adaptation dynamique des applications à base de composants, notamment les travaux liés à des langages de description d'architecture et ceux liés à des modèles de composants. Enfin, dans la section 2.5, nous faisons un bilan de cette étude de l'adaptation des applications à base de composants.

2.1 Composants et applications à base de composants

Différents modèles de composants existent. Même s'ils possèdent parfois des caractéristiques communes, ils diffèrent sur d'autres caractéristiques propres relatives aux composants, à la composition de composants, à leurs cycles de vie, ... Certains modèles sont **hiérarchiques** car ils intègrent la notion de composant *composite*, c'est-à-dire un composant qui contient des sous-composants. Les modèles non hiérarchiques sont dits **plats** car ils n'impliquent que des composants *primitifs*.

Cette section nous permet de définir ce que sont les composants et les applications à base de composants. Dans la sous-section 2.1.1, nous définissons le concept de composant. Et, dans la sous-section 2.1.2, nous explicitons les étapes du cycle de vie des applications à base de composants. Enfin, dans la sous-section 2.1.3, nous discutons des principales forces et limites des modèles de composants actuels.

2.1.1 Caractérisation d'un composant

Il n'existe pas, à ce jour, une définition complète et précise d'un *composant logiciel*. Mais, il existe plusieurs caractérisations d'un composant sur lesquelles la communauté semble s'accorder. La proposition de Szyperski est largement admise au sein de la communauté.

> « Un **composant logiciel** est une unité de composition avec des interfaces spécifiées de manière contractuelle et des dépendances explicites au contexte. Un composant logiciel peut être déployé seul et il peut être composé par d'autres personnes que ses développeurs (des tiers) » [Szy98].

Un composant ressemble beaucoup à un objet car il encapsule des attributs (propriétés configurables) et des fonctions qui caractérisent son comportement. L'*encapsulation* est un des principaux points communs entre les objets et les composants. Par contre, l'*héritage* n'est pas un mécanisme qui apparaît clairement dans la programmation par composants. On peut dire que l'héritage a été substitué par le mécanisme de composition (*i.e.* l'inclusion de composants dans un autre composant), dans la mesure où ces deux mécanismes ont pour objectif de permettre la réutilisation du code pour développer de nouvelles entités (objets ou composants) à partir d'entités déjà présentes. Une autre différence fondamentale entre objets et composants concerne la « visibilité » des interactions entre les différentes parties d'une application. En effet, les composants sont destinés à interopérer entre eux de manière explicite, c'est-à-dire qu'un composant peut utiliser plusieurs composants et il peut aussi être utilisé par d'autres composants. Deux composants ne peuvent interagir que s'ils ont des *interfaces* compatibles entre elles. Par exemple, si les fonctionnalités (généralement appelées services) qui sont nécessaires au premier composant sont décrites dans une interface requise, alors ces mêmes fonctionnalités doivent être décrites chez le second composant sous la forme d'une interface fournie. Lorsqu'on dit qu'un composant "C1" utilise un composant "C2", cela signifie que "C1" peut utiliser un service fourni par "C2". Et, dans certains cas, il est possible que "C1" ne puisse pas fonctionner sans "C2". L'explicitation des dépendances entre les composants facilite, entre autres, l'intégration des composants dans une application car les interactions entre les différentes parties de l'application se trouvent clairement affichées.

Un composant est principalement caractérisé par ses propriétés configurables, ses interfaces, ses contraintes techniques et les modes de coopération avec les autres composants (voir figure 2.1).

Propriétés configurables Les propriétés configurables (*e.g.* état d'un attribut et état de l'activité) sont accessibles à travers des interfaces de

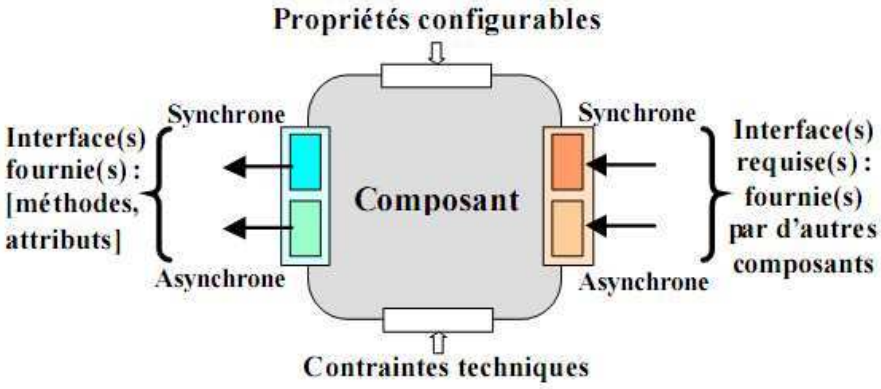

FIGURE 2.1 – Composant logiciel.

contrôle qui permettent d'*inspecter* ou de *(re)configurer* le composant. Typiquement, un composant contient un ensemble d'attributs qui le caractérisent et qui peuvent influer sur son fonctionnement. Par exemple, un composant « Compteur circulaire » disposant d'une propriété configurable « période » peut être utilisé pour compter des heures ou des minutes selon que la valeur de « période » est 24 ou 60.

Interfaces fournies Les interfaces fournies par un composant définissent les opérations qu'un composant met à disposition des autres composants.

Interfaces requises Les interfaces requises par un composant définissent les services (opérations) que ce composant utilise. Ainsi, les interfaces requises et fournies permettent de représenter les dépendances explicites entre deux composants.

Contraintes techniques Les contraintes techniques caractérisent un composant en termes de propriétés extra-fonctionnelles, qui peuvent être : la sécurité, la persistance, les transactions, la qualité de service ou les besoins en ressources matérielles.

Mode de coopération On distingue plusieurs modes de communication entre les composants. Par exemple, lorsqu'on invoque un service chez un composant, on peut utiliser un mode *synchrone* (le composant appelant reste bloqué jusqu'à la réponse de l'appelé) ou alors un mode *asynchrone* (notamment par un système de boîtes aux lettres). On pourrait aussi utiliser un mode de *diffusion continue*, notamment dans une application de diffusion de contenu multimédia sur un réseau à la manière de Layaida & Hagimont [LH05]. A priori, un composant donné peut utiliser plusieurs modes de communication.

Les composants sont aussi explicités par des descripteurs qui sont renseignés tout au long de leur cycle de vie, notamment durant l'assemblage des composants dans le cas des composites.

2.1.2 Cycle de vie d'application à base de composants

Une des particularités de la programmation par composants tient dans le fait que les étapes qui composent le cycle de vie d'une application à base de composants correspondent à celles qui décrivent habituellement le cycle de vie en génie logiciel. Cela signifie qu'on peut réutiliser un certain nombre de techniques ou de principes du génie logiciel pour la programmation par composants [Crn01]. Cependant, on peut distinguer deux cycles imbriqués dans le cas de la programmation par composants : un cycle spécifique au développement de composants et un cycle plus général qui définit des applications à base de composants.

2.1.2.1 Cycle de vie général des applications à base de composants

Les étapes successives de ce cycle sont analogues à celles qui sont définies en génie logiciel (voir la figure 2.2).

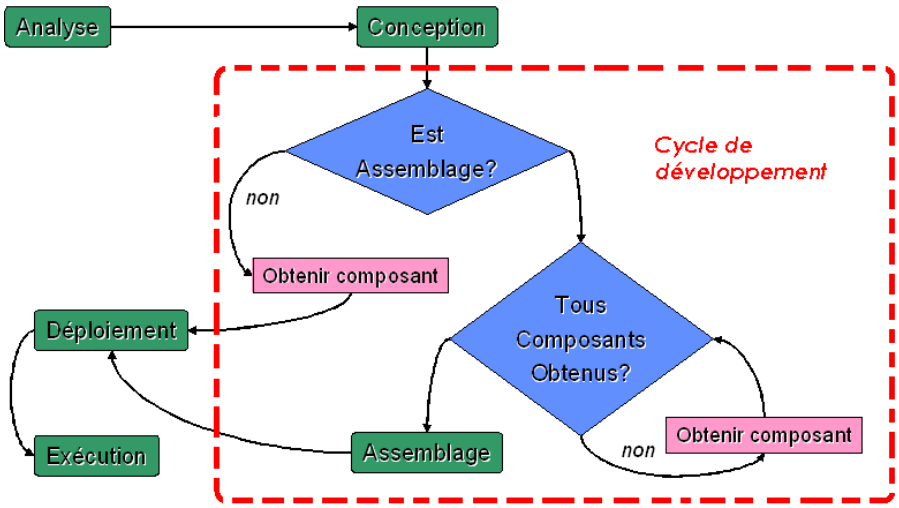

FIGURE 2.2 – Cycle de vie d'une application à base de composants.

Analyse Il faut analyser à la fois les besoins (fonctionnels et non fonctionnels) des composants (niveau composant) et les besoins d'interactions entre les composants (niveau application). Le résultat de l'analyse se représente typiquement par un ensemble de diagrammes de type UML

(« *Unified Modeling Language* »). Les spécifications de l'application sont obtenues par l'*analyste* à partir des informations que lui a fournies le *client* (*i.e.* l'utilisateur) de l'application.

Conception On fixe un modèle de composant (comme Fractal [Obj02, BCS02] ou CCM [GRO02, Obj06]) et on projette les spécifications des composants dans ce modèle. Cette étape est gérée par un *concepteur*.

Cycle de développement d'une application à base de composants Il s'agit de l'étape générale de développement qu'on retrouve en génie logiciel. En programmation par composants, la phase de développement des applications est décomposable en plusieurs étapes que nous explicitons en 2.1.2.2. Cette étape est gérée par un *développeur*.

Déploiement L'étape de déploiement signifie en général le chargement des composants sur une machine, leur initialisation (installation) et leur démarrage. Cette étape est gérée par un *déployeur*.

Exécution L'étape d'exécution signifie qu'un utilisateur administre l'application ainsi déployée et/ou emploie les services fournis par celle-ci. Cette étape est gérée par un ou plusieurs *administrateurs* ou *usagers*.

En général, on associe chacune de ces étapes à des acteurs humains distincts. D'autre part, si, arrivée en phase d'exécution, l'application doit être modifiée (*phase de maintenance*), alors il faut faire un bond en arrière dans le cycle de vie de l'application.

2.1.2.2 Cycle de développement d'une application à base de composants

Le développement d'une application à base de composants est fondé sur un processus cyclique visant à développer des composants, possiblement à partir de composants existants, qui peuvent être utilisés dans beaucoup d'applications (même celles qui n'existent pas encore) [Crn01]. Par conséquent, le cycle de développement des applications à base de composants (figure 2.2) doit idéalement utiliser une ou plusieurs « bibliothèques de composants » où l'on peut *rechercher* des composants afin de les (ré)utiliser. De plus, chaque composant ainsi développé doit être *ajouté* à ces bibliothèques en vue d'une réutilisation ultérieure, éventuellement dans des projets différents. Le mécanisme d'obtention d'un composant est illustré sur la figure 2.3. Dans le cas d'un composite qui n'est pas présent dans la bibliothèque de composants, le mécanisme d'obtention d'un composant s'appelle récursivement pour essayer de récupérer des sous-composants.

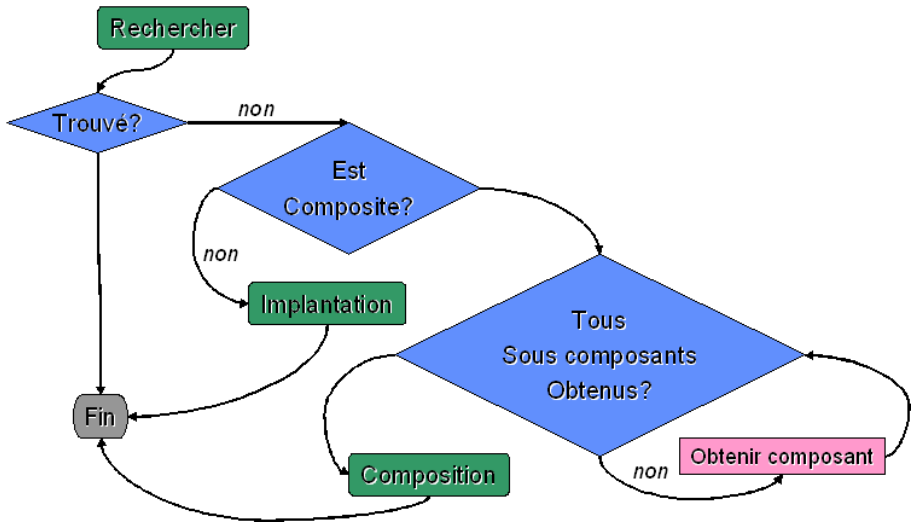

FIGURE 2.3 – Mécanisme d'obtention d'un composant

Ce cycle de développement décrit les étapes permettant de définir des composants *primitifs* (implantés directement), des composants *composites* (composants contenants plusieurs sous-composants primitifs ou composites) et des *assemblages de composants* (ensembles de composants interconnectés).

Implantation Cette étape marque le développement d'un composant dans un langage de programmation conformément au modèle de composants et aux spécifications de ce composant. Le composant peut être configuré à la fin de l'implantation, en fixant des valeurs (par défaut) aux propriétés configurables du composant.

Assemblage L'étape d'assemblage consiste à configurer individuellement un ensemble de composants et à les interconnecter en fonction du comportement global souhaité. Par exemple, on peut considérer deux compteurs circulaires dont on fixe la période à 24 et à 60, avant des les relier entre eux afin d'obtenir une horloge heures-minutes.

Composition La composition ressemble à l'étape d'assemblage, mais aussi à l'étape d'implantation puisqu'elle consiste à créer un composant contenance un assemblage interne. Le processus de création d'un composant composite peut être défini comme suit.

1. Implantation d'un composite vide
2. Insertion des sous-composants
3. Assemblage des sous-composants
4. Importation/Exportation d'interfaces entre le composite et les

sous-composants

Notons finalement qu'une application développée à partir de composants existants peut aussi bien être représentée sous forme d'un assemblage de composants que sous la forme d'un seul composant (composite).

2.1.3 Programmation par composants

La programmation par composants possède plusieurs avantages. Elle favorise à la fois la modularité d'une application grâce à des composants ayant un comportement assez explicite et le plus découplé possible les uns des autres. De plus, elle favorise la réutilisabilité des composants dans des contextes applicatifs différents. Enfin la programmation par composants est mise en œuvre grâce à un processus explicite et faisant intervenir des acteurs distincts, à la manière de ce qui est préconisé en génie logiciel. Par la suite, nous appellerons **développeur d'application** celui qui s'occupe des phases de développement de l'application. Ceux qui s'occupent des phases antérieures et postérieures au développement seront appelés respectivement **concepteur d'application** et **administrateur d'application**.

Cependant, la programmation par composants doit faire face à un certain nombre de défis. Parmi les problèmes notoires des modèles de composants actuels, on peut souligner la nécessité de pouvoir faire interagir des composants issus de modèles différents (*compatibilité des modèles*). De même, certains modèles mélangent des étapes du cycle de vie des applications à base de composants, ce qui oblige différents acteurs du cycle à en maîtriser plusieurs étapes simultanément (*découplage des étapes du cycle*). Enfin, beaucoup d'efforts doivent être fournis afin de faciliter la maintenance et l'évolution des applications (*administration des applications*). Cela passe par l'intégration de la « problématique de l'adaptation » au cœur même du cycle de vie des applications à base de composants.

2.2 Dimensions de l'adaptation logicielle

Aujourd'hui les logiciels répondent à des besoins toujours plus spécifiques et complexes. A cause de cette complexité, il n'est pas rare qu'un logiciel ne corresponde pas totalement à ses spécifications. Il peut aussi arriver que les spécifications initiales d'un logiciel ne soient plus pertinentes après plusieurs mois d'utilisation, ou même pendant sa réalisation. C'est ainsi que s'impose le besoin d'adaptation des logiciels. Nous employons le verbe « adapter » comme « se conformer à des conditions nouvelles ou différentes » (d'après

WordNet [1]). Si le changement des conditions a été prévu alors une étape de reconfiguration du logiciel peut suffire. Par contre, si le changement des conditions n'a pas été prévu, alors il faut adapter le logiciel en en modifiant une partie, voire la totalité. Ces modifications peuvent se révéler très coûteuses à la fois en termes de maintenance et de production du logiciel.

Pour minimiser ces coûts, il convient d'intégrer la problématique d'adaptation dans le cycle de vie du logiciel, en particulier pendant l'étape d'exécution. Ritzau et Andersson ajoutent que « le plus important est de concevoir des systèmes dont les améliorations de maintenance préservent la clarté logique du système, ne serait-ce que pour étendre considérablement la vie du système » [RA00].

La problématique d'adaptation des logiciels n'est pas simple à étudier. Des travaux sur l'adaptabilité des applications [LBB+01, Sen03] ont permis d'identifier différentes dimensions de l'adaptation. Une adaptation peut survenir à différents moments, et pour plusieurs raisons. Elle peut être initiée de différentes façons et peut porter sur des éléments variés d'une application. Ainsi, dans cette sous-section, nous allons répondre aux questions suivantes :

- Dans quels buts adapte-t-on une application ? (section 2.2.1)
- Sur quels éléments de l'application peut porter une adaptation ? (section 2.2.2)
- À quels moments peut-on adapter une application ? (section 2.2.3)
- À qui incombe la décision d'adapter une application ? (section 2.2.4)
- Comment réalise-t-on l'adaptation d'une application ? (section 2.2.5)

Ces dimensions permettent de décrire la nature et la fonction d'une adaptation logicielle. Elles peuvent servir de critères pour comparer le degré d'adaptabilité de différents systèmes.

2.2.1 Motivations

Dans [KBY02], Ketfi et al. explicitent les raisons qui peuvent amener à adapter une application. Pour ce faire, ils définissent quatre types d'adaptations qui se distinguent par des motivations différentes, plutôt que par des mécanismes de mise en œuvre : l'adaptation corrective, l'adaptation adaptative, l'adaptation évolutive et l'adaptation perfective.

- *Adaptation corrective (Correction des bogues)* Pour corriger les erreurs de fonctionnement d'une entité au sein de l'application, il est nécessaire de la modifier ou de la remplacer par une nouvelle version.

1. WordNet : `http://wordnet.princeton.edu`

- *Adaptation adaptative (Flexibilité)* Il s'agit de faire évoluer l'application en fonction d'un éventuel changement de contexte (nouveau système d'exploitation, nouveau composant matériel, intervention d'utilisateur, ...). Par exemple, pour un téléphone portable si l'application détecte un environnement sonore trop bruyant, la fonction vibreur va être activée.
- *Adaptation évolutive (Évolution des fonctionnalités)* C'est la possibilité d'ajouter des fonctionnalités à cause de l'évolution des besoins du client.
- *Adaptation perfective (Performance)* L'objectif est d'optimiser les performances de l'application. Pour ce faire, on peut améliorer le comportement de certaines parties de l'application pour résoudre une tâche plus efficacement.

Notons que ces différentes catégories ne sont que des points de vue et qu'elles ne sont pas exclusives. Par exemple, il est tout à fait possible que l'adaptation corrective d'une application ait un effet bénéfique sur ses performances. Nous donnons dans la section 2.2.3 des exemples concrets motivant des adaptations à différents moments du cycle de vie du logiciel.

2.2.2 Portée

Partant du principe que « un système logiciel est une architecture composée d'un ensemble de boîtes reliées entre elles » [LBB+01], une adaptation peut porter sur une boîte, sur une liaison entre deux boîtes ou bien sur l'architecture entière.

Boîte Une boîte est un terme neutre pour désigner une partie d'un système sans présupposer de sa représentation et de son implémentation. Il peut s'agir d'une fonction, d'une classe, d'un composant, d'une entité modulaire quelconque. L'adaptation d'une boîte peut aller d'un simple paramétrage à une transformation complète

Liaison Une liaison établit une relation entre deux boîtes, par exemple un lien d'héritage en programmation orientée objet. L'adaptation d'une liaison consiste à modifier la nature de la liaison (par exemple, un lien de délégation à la place d'un lien d'héritage), ou à appliquer cette liaison sur autre couple de boîtes.

Architecture logique L'adaptation de l'architecture logique d'un système consiste à ajouter, supprimer ou encore remplacer une boîte ou une liaison.

Architecture physique L'adaptation de l'architecture physique d'un système s'appelle un re-déploiement. Cela consiste à modifier la répartition d'un système logiciel sur un environnement d'exécution matériel com-

posé de un ou plusieurs sites de déploiement. En général, il s'agit donc de migrer des boîtes du système sur un site différent.

2.2.3 Moments

Les applications peuvent être adaptées à plusieurs moments de leur cycle de vie.

2.2.3.1 Trois moments d'adaptations

Trois moments d'adaptation recouvrent toutes les étapes du cycle de vie d'une application, depuis la conception jusqu'à l'exécution.

- *Adaptation statique :* L'adaptation statique concerne les étapes du cycle qui sont avant le lancement de l'application, c'est-à-dire l'analyse, la conception et le développement (*i.e.* l'implantation).
- *Adaptation semi-dynamique :* L'adaptation semi-dynamique désigne l'adaptation qui peut apparaître à l'étape de déploiement de l'application. Dans ce cas, on s'intéresse à la dynamicité du processus d'installation de l'application. Des opérations d'adaptation peuvent avoir lieu pendant que le processus d'installation de l'application est en cours. En effet, lorsque le contexte de déploiement d'une application n'est pas totalement connu à l'avance, il doit être découvert dynamiquement au moment du déploiement, comme l'illustre Senart [Sen03].
- *Adaptation dynamique :* L'adaptation dynamique intervient lors de la phase d'exécution de l'application. Les modifications dynamiques permettent de satisfaire les besoins les plus tardifs, qu'ils soient anticipés ou non.

2.2.3.2 Illustration du besoin d'adaptations à différents moments

Nous donnons quelques exemples qui illustrent le besoin d'adaptation à ces différents moments du cycle de vie d'une application.

- *Adaptation statique :*
  - Les modifications des spécifications d'un composant qui est en cours de développement engendrent une adaptation statique de son implantation.
- *Adaptation semi-dynamique :*
  - Toutes les conditions de déploiement ne peuvent pas être prévues de manière exacte et parfois le concepteur préfère laisser un certain degré de liberté au déployeur (généralement c'est l'administrateur de l'application), à l'instar des applications qui demandent qu'on

spécifie un nouveau répertoire d'installation si le répertoire par défaut ne convenait pas.
  - Un déploiement entièrement automatisé peut nécessiter de configurer l'application afin qu'elle puisse s'accommoder d'un environnement d'exécution particulier (par exemple, pour définir le caractère de fin de ligne d'un composant d'édition de texte par CR sous Macintosh et par LF sous Unix).
- *Adaptation dynamique :*
  - Lors d'un changement de contexte d'exécution comme une trop faible quantité de batterie sur un téléphone mobile, il peut être souhaitable de renoncer à certains services facultatifs pour économiser de l'énergie ; l'adaptation consiste alors à réorganiser l'architecture logique de l'application pour minimiser les aspects gourmands en ressources (par exemple, passer d'un affichage 3D à un affichage 2D). C'est un exemple d'adaptation anticipée.
  - Un utilisateur/administrateur peut vouloir réimplanter une partie d'une application pour corriger un problème de fonctionnement ou encore pour améliorer les performances de traitement et par exemple, on doit pouvoir modifier un algorithme de compression vidéo lorsque le besoin s'en fait sentir dans une application de diffusion vidéo, et ceci sans avoir à stopper la diffusion. La correction d'un problème est un exemple d'adaptation non anticipée.

En général, lorsque des besoins d'adaptation apparaissent tardivement dans le cycle de vie d'une application, c'est que leur cause n'a pas pu être anticipée. Ce n'est pas l'aspect tardif des adaptations qui pose le plus de problèmes, mais c'est surtout les contraintes posées par la dynamicité.

2.2.3.3 Difficultés dues à la dynamicité

Pour réaliser l'adaptation, il faut généralement stopper l'application. Cependant, il existe des applications critiques qui doivent s'exécuter de manière continue car elles doivent être disponibles à tout moment (applications du domaine médical, bancaire, télécoms, gestion de production, ...). Dans ce cas, il faut envisager d'adapter dynamiquement ces applications.

Deux autres cas, aussi pertinents, sont cités par Kefti et al. [KBC02b] :
- le cas des applications dont l'environnement d'exécution change constamment, ce qui rend inadapté l'arrêt de l'application à chaque modification,
- les applications dont l'arrêt est coûteux (en termes de ressources, de fonctionnalités, etc) pour les entreprises.

L'adaptation dynamique est une solution (pour corriger des bugs, faire évoluer les fonctionnalités d'une application, ...), mais qui pose des défis. Il est difficile de déterminer les adaptations nécessaires et de les réaliser correctement. De plus, les applications qui s'exécutent continuellement doivent souvent respecter un certain degré de disponibilité. Une opération de mise à jour ne doit pas remettre en cause la qualité de service qu'on est en droit d'attendre dans certaines applications. Ainsi, l'effort à faire pour optimiser les opérations d'adaptation est d'autant plus grand lorsqu'on se place dans des systèmes à fortes contraintes comme les applications temps-réel (limitées en temps d'exécution) et les applications embarquées sur des dispositifs limités en ressources matérielles (CPU, mémoire et énergie).

2.2.4 Acteurs

L'adaptation d'applications est un processus qui se déroule en plusieurs étapes. Chacune de ses étapes fait intervenir des acteurs particuliers, qu'ils soient humains ou logiciels. L'adaptation d'une application n'est pas forcément la responsabilité de l'administrateur de cette application. En particulier, si son concepteur l'a anticipé, l'application peut initier sa propre adaptation automatiquement, *i.e.* sans intervention humaine.

2.2.4.1 Intervenants humains

Hormis l'utilisateur, les principaux acteurs du cycle de vie d'un logiciel sont le concepteur, le développeur et l'administrateur (voir le paragraphe 2.1.3). Comme le besoin d'adaptation peut se faire sentir à tout moment, chaque acteur du génie logiciel est potentiellement l'initiateur d'une opération d'adaptation.

2.2.4.2 Déclenchements automatiques

L'initiateur d'une opération d'adaptation n'est pas toujours un intervenant humain. Des entités logicielles (des composants par exemple) peuvent également jouer ce rôle d'initiateur d'adaptation lorsque les critères de déclenchement de l'adaptation sont clairement identifiés et par conséquent facilement automatisables.

Dans certains types d'applications dites « sensibles au contexte » (« *context-aware* »), des adaptations peuvent être déclenchées par des capteurs physiques ou logiciels mesurant par exemple la température ambiante ou la position géographique [DSA01]. On peut imaginer notamment un système qui

détermine un trajet pour une visite guidée ou encore un système de régulation de la consommation d'énergie dans une maison. C'est un cas typique où l'initiateur d'une opération d'adaptation est à l'intérieur même du logiciel.

L'adaptation joue également un rôle prépondérant dans les *systèmes autonomiques* ou auto-adaptatifs [KC03, NCCR05]. Ces systèmes autonomiques étant directement inspirés des Systèmes Multi-Agents (SMAs), ils se composent de différentes entités logicielles dont le fonctionnement dépend presque exclusivement des autres entités, afin de minimiser les interventions humaines. C'est un cas typique où l'initiateur d'une opération d'adaptation est une entité logicielle externe.

2.2.5 Mise en œuvre de l'adaptation

Pour comprendre comment mettre en œuvre l'adaptation d'applications, il nous faut décrire les étapes du processus d'adaptation ainsi que les approches qui sont utilisées. Ces approches se distinguent non seulement par rapport aux techniques utilisées mais aussi par la qualité du processus d'adaptation compte-tenu des exigences de l'application ou du domaine d'application. C'est pourquoi, certains travaux mettent l'accent sur les garanties qu'ils obtiennent en contrôlant rigoureusement le processus d'adaptation.

2.2.5.1 Techniques de l'adaptation

L'adaptation de logiciels peut se faire en suivant différentes techniques. Parmi les approches basiques (*i.e.* bas niveau), on distingue notamment le *re-paramètrage* (ou reconfiguration) dans le cas d'adaptations anticipées et la *transformation de code* portant sur le code source ou sur du « *bytecode* » (comme le font Hnetynka et Tuma [HT03]) dans le cas des adaptations non anticipées. D'autres approches possibles permettent l'adaptation à un haut niveau. C'est notamment le cas de la réflexion, de la programmation par aspects et des design patterns.

La *réflexion* [Smi82] est une technique permettant à un logiciel de s'auto-représenter et de s'auto-manipuler. Ce mécanisme est utilisé pour programmer de manière générique en manipulant le code de base d'une application. Par exemple, la réflexion a été mise en œuvre dans la thèse de Rank [Ran02] pour définir une architecture qui prend en charge dynamiquement l'évolution de logiciels.

La *programmation par aspects (AOP)* [KLM+97, FECA05] est un mécanisme permettant de réaliser des opérations transversales sur le code

métier d'une application ou sur son code technique (la distribution, la persistance, ...). L'AOP a été utilisée pour faire de l'adaptation évolutive de logiciels [LRZJ04] et pour l'adaptation d'applications à base de composants [VS04].

Les *patrons de conception* (« *Design Patterns* » [GHJV95]) sont des solutions récurrentes aux problèmes de conception du génie logiciel orienté-objet. Les patrons « *Bridge* » (qui sépare l'interface d'un objet de son implémentation) et « *Strategy* » (qui encapsule un algorithme dans une classe) sont utilisés pour adapter respectivement la structure et le comportement d'un objet.

2.2.5.2 Contrôle de l'adaptation

Même si les dimensions de l'adaptation sont désormais bien connues, la réalisation concrète des adaptations pose encore des problèmes. L'étape de réalisation de l'adaptation doit être contrôlée pour que l'adaptation soit *sûre*, c'est-à-dire que le programme maintient son intégrité durant une adaptation [ZYCM04]. Il s'agit notamment de :

- garantir que les opérations d'adaptation vont toujours réussir,
- garantir que la réalisation de l'adaptation correspond exactement à ce qui a été prescrit dans l'étape de décision, ou encore
- garantir que les opérations d'adaptation ne causent pas des effets indésirables sur le fonctionnement de l'application.

2.2.5.3 Étapes de l'adaptation

Une adaptation se déroule en trois étapes successives :

- le *déclenchement* qui permet de n'autoriser les adaptations que si certaines conditions sont satisfaites et qui implique un initiateur et un moment d'adaptation,
- la *décision* qui consiste à déterminer les modifications à réaliser en fonction des changements subis et qui concerne un initiateur, un sujet, un type, une technique, une stratégie d'adaptation,
- la *réalisation* qui est une mise en pratique de la décision prise précédemment et qui veille à opérer l'adaptation en respectant les règles fixées dans l'étape de décision.

Dans le cas d'une adaptation non anticipable par l'application, mais initiée par un de ses utilisateurs ou administrateurs (par exemple, une correction logicielle permettant de prévenir un bogue), les étapes de déclenche-

ment et de décision sont effectuées au niveau de cet intervenant humain et n'apparaissent pas clairement au niveau logiciel, contrairement à l'étape de réalisation. C'est pourquoi, la distinction des étapes de l'adaptation n'est pas considérée comme primordiale en général. Par contre, ces trois étapes prennent tout leur sens lorsqu'on souhaite *automatiser le processus d'adaptation*. Ce processus s'exprime de manière plus ou moins complexe, selon le comportement désiré pour l'application.

2.2.6 Synthèse

Dans cette section, nous avons traité de l'adaptation d'applications. Une adaptation logicielle peut se décrire selon plusieurs dimensions. Et, à partir des dimensions de l'adaptation, il est possible de définir des critères pour comparer le degré d'adaptabilité de différents systèmes logiciels. Ces critères sont donnés par le tableau 2.1. Notons par ailleurs que les dimensions de l'adaptation ne sont pas orthogonales. Par exemple, une adaptation qui porte sur l'architecture physique de l'application correspond en principe au moment « semi-dynamique ».

Critère	**Valeurs/Éléments**
Motivation (Pourquoi)	Corrective, Adaptative, Evolutive, Perfective
Portée (Quoi)	Boîte, Liaison, Architecture logique, Architecture physique
Moment (Quand)	Statique, Semi-dynamique, Dynamique
Acteur (Qui)	Humain, Logiciel externe, Logiciel interne
Mise en œuvre (Comment)	Technique(Réflexion, AOP, Design patterns) Contrôle(Terminaison, Correction, Sans effet indésirable)

TABLE 2.1 – Critères généraux de l'adaptation

2.3 Adaptation d'Applications à Base de Composants (ABCs)

Ayant vu la problématique d'adaptation logicielle, considérons maintenant le cas des applications à base de composants. Nous nous plaçons dans le cadre du paradigme de programmation par composants. C'est pourquoi, nous explicitons les différentes dimensions de l'adaptation dans le cas d'une Application à Base de Composants (ABCs).

2.3.1 Motivations de l'adaptation d'ABCs

Les principales raisons pour adapter une application à base de composant sont celles déjà évoquées pour l'adaptation d'applications en général (cf. section 2.2.1) : l'adaptation corrective, l'adaptation adaptative, l'adaptation évolutive et l'adaptation perfective. Il y a d'autres besoins qui ne sont pas de l'ordre de la maintenance, mais plutôt de l'amélioration de la qualité de l'application. Il existe une dizaine de critères de qualité dans le domaine du génie logiciel [WBLS01]. Parmi les critères de qualité les plus désirés dans la programmation par composants, on trouve l'interopérabilité [BCP04, BBC02, Ber01, VHT00] et la réutilisabilité [BTV04]. Nous les définissons de la manière suivante :

- *Adaptation pour l'interopérabilité (Standardisation)* Ce type d'adaptation est généralement utilisé lors de l'assemblage et le déploiement, pour rendre un composant compatible avec un grand nombre de contextes (système d'exploitation, interface homme-machine, ...).
- *Adaptation pour la réutilisabilité (Généricité)* L'objectif est d'obtenir des composants mieux réutilisables après l'adaptation, pour que le fonctionnement d'un composant soit le plus découplé possible de son contexte d'utilisation (composants voisins).

À la différence des quatre premières catégories, ces deux dernières se focalisent moins sur les besoins applicatifs d'un logiciel que sur les propriétés des composants utilisés. Elles tendent à améliorer l'interopérabilité et la réutilisabilité d'un composant pour un grand nombre de contextes.

2.3.2 Portée de l'adaptation d'ABCs

Une application à base de composants se caractérise en premier lieu par son architecture (physique et logique). Elle est également caractérisée par l'ensemble des composants qui la composent, c'est-à-dire les boîtes d'après la terminologie de la sous-section 2.2.2. L'adaptation d'une application à base de composants peut avoir lieu au niveau architecture ou au niveau composant. L'impact d'une adaptation sur l'architecture d'une application à base de composants varie en fonction de la cible de l'opération d'adaptation (un composant, une interface, une connexion, etc.). Dans la littérature, la cible d'une opération est appelée **sujet de l'adaptation**, et on parle parfois de **types d'adaptation** pour signifier que l'impact d'une adaptation peut apparaître à différents niveaux (implantation interne à un composant, architecture logique de l'application et répartition physique des composants) [Sen03]. Nous décrivons ci-dessous la portée des adaptations dans les applications à base de composant en mêlant les sujets et les types d'adaptations possibles.

- *L'adaptation de l'**architecture physique** de l'application* consiste à migrer des composants vers d'autres sites d'exécution. Cette opération correspond à un redéploiement de l'application.
- *L'adaptation de l'**architecture logique** de l'application* consiste à ajouter, supprimer ou remplacer des composants. Il s'agit aussi de modifier les connexions entre composants existants. Cette opération correspond à un réassemblage de l'application.
- *L'adaptation des **composants** de l'application* consiste à modifier des composants à plusieurs niveaux (son comportement, son contenu, sa description, etc).
  - *Reconfiguration du composant :* cela correspond à reparamétrer les propriétés configurables d'un composant au travers d'une interface de contrôle.
  - *Réimplantation du composant primitif :* on peut réimplanter la partie interne et/ou externe d'un composant.
    - Réimplantation interne du composant : il peut s'agir de changer une structure de données dans le code, de changer l'implémentation d'une méthode (changement d'algorithme), etc.
    - Réimplantation de la structure externe du composant : il peut s'agir de fusionner des interfaces (par exemple pour rassembler l'ensemble des opérations fournies dans une seule interface) ou inversement de les éclater en plusieurs interfaces ; il est également possible d'ajouter ou de supprimer des services fournis (ou requis).
  - *Réimplantation d'un composant composite :*
    - Réimplantation interne du composant (pour les modèles où les composites ne sont pas vides, *i.e.* les modèles dans lesquels les composites peuvent contenir du code, autrement que par les sous-composants qu'ils contiennent).
    - Réimplantation de la structure externe du composant.
    - Réorganisation interne du composite : cela consiste à ajouter, supprimer ou réassembler des sous-composants ; en un mot, il s'agit de recomposer un composite [2].

Ces types de modifications peuvent être couplés. Par exemple, si on éclate une interface de composant en plusieurs interfaces différentes, il faudra le reconnecter différemment avec les autres composants et il sera peut-être né-

2. Notons que certains auteurs (comme Sénart [Sen03]) préfèrent dire que les réorganisations internes de composants composites sont avant tout des adaptations de l'architecture logique de l'application. Notre choix nous permet d'être cohérent avec les définitions précédentes (voir le paragraphe 2.1.2.2) : (i) l'étape d'assemblage produit un assemblage de composants tandis que (ii) l'étape de composition produit un composant composite.

cessaire de supprimer certains composants devenus inutiles.

2.3.3 Moments de l'adaptation d'ABCs

Les applications à base de composants peuvent être adaptées à plusieurs moments de leur cycle de vie. Comme dans la section 2.2.3, nous distinguons l'adaptation statique, l'adaptation semi-dynamique (*i.e.* au déploiement) et l'adaptation dynamique (*i.e.* en cours d'exécution).

2.3.4 Acteurs de l'adaptation d'ABCs

L'initiateur de l'adaptation d'une application à base de composants peut être humain dans le cas d'une adaptation manuelle ou non dans le cas d'une adaptation automatique. En effet, dans le cas d'une adaptation automatique, l'adaptation est déclenchée à cause d'une entité logicielle externe ou interne à l'application. Lorsqu'il s'agit d'une entité interne, cela peut être (a) un composant particulier (par exemple, un gestionnaire d'adaptation) [LPH04, MG03], (b) une catégorie particulière de composants (notamment des fabriques de composants conçues selon le patron de conception « *Factory* ») [BHP06] ou (c) des composants applicatifs quelconques [BMP06]. Le cas (a) possède l'avantage d'appliquer des conditions de déclenchement communes à tous les composants. Les cas (b) et (c) adoptent une approche distribuée qui est plus difficile à contrôler (notamment pour assurer la cohérence de l'application), mais (b) favorise la séparation des préoccupations entre le code applicatif et le code d'adaptation parmi les composants tandis que (c) est une approche plutôt ad hoc.

Dans le cas d'une adaptation manuelle, plusieurs personnes peuvent déclencher une adaptation. Comme le besoin d'adaptation peut se faire sentir à tout moment, chaque acteur de la programmation par composants est potentiellement l'initiateur d'une opération d'adaptation : un concepteur d'une application à base de composants, un développeur ou un administrateur.

2.3.5 Mise en œuvre de l'adaptation d'ABCs

La problématique du réassemblage de composants logiciels est au cœur de la programmation par composants. C'est pourquoi, dans cette thèse, nous nous focalisons sur la mise en œuvre de l'adaptation au niveau de l'architecture logique de l'application. L'adaptation d'un composant primitif est tout à fait similaire à l'adaptation d'une application telle que nous l'avons décrite dans la section 2.2. De plus, l'adaptation d'un composant composite peut se

ramener à l'adaptation d'une application à base de composants en considérant qu'un composite n'est qu'une enveloppe qui contient un assemblage de composants internes qui peuvent être primitifs ou composites. Enfin, le redéploiement de composants d'une application pose des problèmes spécifiques liés à la distribution et dépend de l'infrastructure de déploiement utilisé, ce qui dépasse le cadre de cette thèse.

Nous utilisons le mot « restructurations[3] » pour caractériser les opérations d'adaptation qui ont un impact sur l'architecture de l'application, comme l'ajout et la suppression de composants. Les restructurations d'applications à base de composants ne visent pas à modifier les composants qui la composent (*e.g.* l'implantation des interfaces d'un composant) sauf lorsqu'il s'agit d'une reconfiguration basée sur une interface extra-fonctionnelle (*e.g.* changement des valeurs de ses propriétés configurables). Lorsqu'un composant ne convient plus à une application, il est supprimé ou remplacé par un composant approprié.

Pour réaliser une restructuration dynamique dans une application à base de composants, il faut opérer des modifications spécifiques. Les opérations de restructuration que nous décrivons concernent l'ajout, la suppression, le remplacement de composants, la reconfiguration des connexions entre composants, etc. Nous décrirons ces opérations, mais sans détailler les mécanismes qui sont mis en œuvre dans les systèmes existants.

2.3.6 Opérations d'adaptation d'ABCs

Les opérations de restructuration permettent de faire évoluer la *topologie de l'architecture applicative, si possible de manière incrémentale*. Cet aspect incrémental signifie qu'il n'y a qu'une partie de l'architecture de l'application qui est modifiée à chaque opération de restructuration. La granularité d'une restructuration est variable dans le sens où une telle adaptation peut parfois porter sur plusieurs composants d'un coup. Enfin, il est clair que la transformation d'une architecture d'application à une autre fait intervenir une succession d'opérations de restructuration. En d'autres termes, il faudra en général définir des opérations de restructuration complexes à partir des opérations de base qui sont disponibles.

2.3.6.1 Opérations simples

Nous pouvons distinguer *quatre genres d'opérations* de restructuration : les insertions, les destructions, les substitutions et les reconfigurations.

3. Dans la littérature, les termes recomposition, reconfiguration, ou plus simplement ré-assemblage sont parfois utilisés pour désigner des modifications plus ou moins similaires.

- *Insertion* pour ajouter des composants
- *Destruction* pour supprimer des composants
- *Substitution* pour remplacer des composants
- *Reconfiguration* pour modifier la façon dont les composants de l'application sont assemblés mais en gardant les mêmes composants. Nous employons le terme de reconfiguration de l'application pour désigner de manière générique le fait de modifier la valeur des propriétés configurables d'un composant d'une part, et les connexions entre différents composants de l'application d'autre part.

Ces opérations abstraites se matérialisent différemment dans les systèmes existants. Mais, il existe des étapes qui se retrouvent dans la plupart d'entre eux. Nous pouvons les discuter sans perte de généralité.

- *Activation et désactivation* Les mécanismes d'activation et de désactivation d'un composant visent respectivement à autoriser et interdire l'accès aux opérations fournies par ce composant. Ces mécanismes sont utilisés dans certains systèmes pour garantir un certain degré de stabilité de l'application durant la restructuration.
- *Chargement et déchargement* Le chargement d'un composant consiste à intégrer un composant extérieur à l'application (par exemple, depuis une bibliothèque de composants) de sorte qu'il devienne un élément de l'application. Le déchargement d'un composant implique de supprimer toutes les instances/occurrences de ce composants dans l'application.
- *Connexion et déconnexion* Les mécanismes de connexion et de déconnexion sont à la base de l'étape d'assemblage d'une application à base de composants. Elles régissent les dépendances entre les composants qui fournissent des services et ceux qui les utilisent.
- *Configuration* La configuration d'un composant consiste à le paramétrer grâce à son interface de contrôle, *i.e.* modifier la valeur de ses propriétés configurables.
- *Capture et restauration* Il s'agit de mécanismes spécifiques aux opérations de substitution. Par exemple, pour remplacer un composant par un autre, il faut transférer les états internes (valeurs des propriétés configurables) et externes (dépendances explicites envers les autres composants, à travers des interfaces) de l'ancien composant vers le nouveau. D'où l'utilité des mécanismes de capture et de restauration d'état.

Les systèmes adaptables implantent ce genre de mécanismes [4] pour réaliser des opérations de restructuration étape par étape. Un exemple typique d'opération de substitution basée sur ces étapes est décrit dans la thèse de Sénart [Sen03]. Nous établissons la correspondance entre ces étapes et les mécanismes ci-dessus.

Remplacement d'un composant par un autre :

1. arrêt de l'exécution de l'instance à remplacer (désactivation)
2. capture de l'état interne du composant (capture)
3. remplacement effectif du composant (chargement du nouveau et déchargement de l'ancien)
4. restauration de l'état dans le nouveau composant (restauration)
5. démarrage du nouveau composant (activation)

Nous pensons que les opérations d'insertion et de destruction posent moins de problèmes que les opérations de substitution quant au fonctionnement sûr de l'application. En effet, le remplacement d'un composant peut nécessiter d'agir sur le flot d'exécution de l'application, notamment en suspendant des tâches en cours chez le composant à remplacer ou chez ceux qui utilisent ses services fournis. *Un composant à remplacer est à la fois utile (contrairement au composant à supprimer) et utilisé (contrairement au composant à ajouter).*

Nous avons décrit les principaux mécanismes qui composent les opérations de restructuration d'une application à base de composants. Dans le domaine des architectures logicielles, les quatre opérations fondamentales de restructuration sont l'ajout et la suppression de composants et de connexions [WLF01]. Bien que leurs noms soient variables, la plupart des langages de reconfiguration fournissent ces quatre opérations [KM90, Med96, BCDW04].

2.3.6.2 Opérations complexes

Le choix des opérations de restructuration est important. Nous pouvons nous interroger sur le nombre d'opérations différentes à définir. Medvidovic définit les opérations pour ajouter un composant, supprimer un composant et mettre à jour un composant (le remplacer par un autre) dans le style ar-

4. Chacune de ces sous-opérations est supposée être sûre, étant donné un modèle de composants qui les propose. Donc, nous partons du principe que les opérations de restructuration fournies par un modèle de composants maintiennent la cohérence de l'application.

chitectural[5] C2 [Med96]. Grâce aux fonctions de connexion/déconnexion de composants et de connecteurs, il est possible de changer complètement d'architecture. Lorsque la reconfiguration de l'architecture d'une application fait intervenir trop d'opérations, Medvidovic conseille de spécifier directement la nouvelle architecture. Cependant, cette façon de faire ne semble pas réaliste si nous voulons que la restructuration se fasse dynamiquement. Idéalement, il faut pouvoir exprimer des opérations de restructuration plus complexes.

Comme la programmation par composants vise également à développer directement des assemblages de composants (voir la figure 2.2), il semble approprié de définir des opérations de restructuration de base qui s'appliquent également aux assemblages de composants. Par exemple, des *opérations qui visent à remplacer un composant par un assemblage de composants* ou l'inverse sont très utiles. Aussi, il est souhaitable d'avoir des opérations de restructuration qui impliquent plus que deux composants à la fois.

Une autre capacité intéressante des opérations de restructuration consiste à permettre de les *composer à la demande*, par exemple en déclarant plusieurs adaptations[6] successives. La plupart des approches pour définir des architectures logicielles dynamiques prennent en charge certaines opérations composées d'adaptation [BCDW04]. Certaines d'entres elles comme CommUnity [WLF01] et Gerel [EW92] utilisent plusieurs sortes d'opérateurs pour composer des opérations d'adaptation entre elles (les séquences, les choix, les itérations).

2.3.7 Qualité des opérations d'adaptations d'ABCs

Les opérations de restructuration sont généralement le résultat de la composition de plusieurs autres opérations de restructuration plus simples, de manière à pouvoir exprimer toutes sortes de modifications. Ainsi, ces opérations permettent de restructurer une application à base de composants. Néanmoins, nous pouvons envisager d'utiliser des opérations bien plus élaborées et spécifiques. En effet, il est primordial pour un concepteur d'applications de contrôler parfaitement le déroulement des adaptations de son application. Pour ce faire, il doit pouvoir spécialiser facilement les opérations d'adaptations dont il dispose. Par conséquent, la qualité des opérations d'adaptations visant les applications à base de composants doit pouvoir s'évaluer selon des critères bien précis. En étudiant de nombreux travaux, notamment les avan-

5. « Les styles architecturaux sont des ensembles réutilisables de décisions de conception et de contraintes qui sont appliquées à une architecture pour induire certaines qualités désirées. » [IfSR]

6. Nous supposons qu'une composition d'opérations de restructurations sûres est elle-même sûre.

tages et les inconvénients que leurs auteurs mettent en exergue, nous avons identifié les cinq critères qui suivent.

2.3.7.1 Cohérence des adaptations

Dans [Ket04], Ketfi qualifie une restructuration de cohérente si elle garantit au moins le passage de l'application à restructurer d'un *état stable* vers un autre. L'état de l'application est qualifié de stable lorsque l'application est capable de continuer normalement son exécution à partir de celui-ci, et de fournir des résultats attendus. Le maintien de cohérence passe généralement par des mécanismes de transferts d'état de l'application et de coordination des actions en cours.

Par ailleurs, les opérations de restructuration doivent nécessairement *se terminer*. Elles doivent *limiter les effets indésirables* selon les motivations de l'adaptation. Par exemple, dans le cas d'une adaptation perfective, on souhaite absolument optimiser les performances de l'application après l'adaptation, alors que dans le cas d'une adaptation corrective, on souhaite corriger les erreurs de l'application quitte à ce qu'elle soit un peu moins performantes. Enfin, le point essentiel est de pouvoir assurer une *exécution correcte* de l'application d'une part, et des adaptations d'autre part. Ici, le terme « correcte » caractérise simplement l'adéquation entre les spécifications du concepteur de l'application, leurs évolutions éventuelles (en cas d'adaptation évolutive), l'intention de l'initiateur de l'adaptation (autres cas d'adaptation) et ce qui est exécuté dans l'application.

Ainsi, nous définissons la **cohérence** comme l'agrégation des trois propriétés que sont la correction, la terminaison et l'absence d'effets indésirables. En termes simples, une opération d'adaptation est qualifiée de cohérente lorsqu'elle fait exactement ce qui est attendu d'elle.

2.3.7.2 Performance des opérations d'adaptation

Certaines applications sont susceptibles d'être adaptées fréquemment au cours de leur cycle de vie, c'est pourquoi on peut entreprendre de réduire le coût (en temps et en ressources) des opérations d'adaptations. Nous définissons une opération d'adaptation comme **performante** lorsque son coût en temps d'exécution (performance absolue) ou en ressources matérielles consommées (performance relative) est bas.

L'optimisation des performances des opérations d'adaptations est un problème technique souvent lié à une application et une implantation particulière. Malgré tout, il est possible d'appréhender le problème d'optimisation des restructurations d'un point de vue stratégique, en imposant des

contraintes sur l'ordonnancement (c'est-à-dire l'ordre et l'instant de déclenchement) des opérations qui servent aux restructurations afin d'en contrôler l'exécution.

2.3.7.3 Disponibilité de l'application adaptée

Il faut souvent *garantir la disponibilité de certaines fonctionnalités de l'application.* Cela permet d'assurer une certaine qualité de service dans l'application. Ainsi, nous disons qu'une opération d'adaptation favorise la **disponibilité de service** lorsqu'elle gère le maintien des performances[7] de toute l'application ou le maintien de certaines fonctionnalités (généralement essentielles) de l'application.

La *qualité de service* d'une application à base de composants est un critère important dans l'exécution d'une application. Favoriser la disponibilité de certaines fonctionnalités peut induire des impacts négatifs sur d'autres fonctionnalités, et donc sur la cohérence de l'application. Dans ce cas, il faut rendre indisponibles les fonctionnalités qui ne sont plus assez performantes. On peut aussi choisir d'interdire les opérations d'adaptations qui pourraient bloquer ou perturber un assez grand nombre de composants.

Enfin, certains modèles de composants permettent de faire co-exister plusieurs versions d'un composant pour des raisons de disponibilité de services ou de performance. Par exemple, dans [Ajm04], un mécanisme est prévu pour que les anciennes versions d'un composant soient supprimées quand elles ne sont plus utiles. Ce genre de mécanisme global n'est que rarement fourni par les infrastructures d'adaptations car la gestion de versions de composants pose des problèmes multiples qui sont répertoriés dans [Bra03].

2.3.7.4 Simultanéité des adaptations

Lorsqu'il y a plusieurs *adaptations simultanées*, il faut les coordonner, c'est-à-dire gérer l'ordonnancement des adaptations :

- pour des besoins évidents de correction si les opérations d'adaptations sont concurrentes (si elles concernent les mêmes composants),
- pour d'autres besoins comme le maintien de la qualité de services si les opérations d'adaptations sont parallèles (*i.e.* non concurrentes).

Nous disons qu'une infrastructure d'adaptation gère la **simultanéité** des adaptations lorsqu'elle permet d'adapter à la fois plusieurs composants ou connexions.

7. Attention à ne pas confondre les performances de l'application avec la performance des adaptations tels qu'on la définit dans le paragraphe 2.3.7.2.

Il s'agit de prendre en charge des adaptations complexes, c'est-à-dire composées de plusieurs insertions, destructions, substitutions et reconfigurations (voir le paragraphe 2.3.6). En maîtrisant l'ordonnancement (l'ordre et l'instant) d'exécution des adaptations, on contrôle sur la qualité ou l'efficacité de l'adaptation et de l'exécution de l'application. Cela revient à associer en quelque sorte des stratégies (par exemple, liées à des critères de qualités de services) à nos opérations d'adaptation. Ces stratégies sont d'autant plus intéressantes lorsque les adaptations s'opèrent automatiquement.

2.3.7.5 Ouverture de l'infrastructure d'adaptation

Bien souvent, il n'est pas possible de satisfaire tous les critères de qualité des opérations d'adaptation. C'est pourquoi, il faut établir un compromis entre ces critères notamment dans le cas des adaptations dynamiques. Pour qu'une opération de restructuration soit vraiment dynamique, elle doit être non bloquante pour l'exécution de l'application à base de composants. Et, puisqu'il ne faut pas bloquer toute l'application, il faut pouvoir déclencher ces opérations aux bons moments et aussi les exécuter selon un ordre approprié. Cela se justifie par le fait que les *opérations de restructuration sont possiblement complexes*. Cette complexité est d'origine diverse :

- il peut y avoir beaucoup de contraintes à respecter (cf. cohérence),
- il peut y avoir beaucoup de composants impliqués dans l'opération (cf. performance),
- il peut y avoir beaucoup de fonctionnalités à préserver (cf. disponibilité),
- il peut y avoir beaucoup de modifications à faire (cf. coordination).

En fait, ces critères peuvent être vus comme des contraintes sur les opérations d'adaptation. Cela suggère la possibilité d'associer différentes stratégies d'adaptation à nos opérations, car une stratégie unique ne pourrait satisfaire complètement tous les critères. Par conséquent, chaque stratégie est un compromis entre les différents critères de qualité des opération d'adaptation. Ainsi, les choix stratégiques s'appliquant à une infrastructure d'adaptation peuvent être exprimés à travers une *politique d'adaptation*. On dit qu'une infrastructure d'adaptation est **ouverte** lorsque les mécanismes et les stratégies d'adaptation qu'elle met en œuvre sont explicites et paramétrables. Par exemple, dans sa thèse [Ajm04], Ajmani utilise le concept de fonctions d'ordonnancement (« *Scheduling Functions* ») pour exprimer différentes stratégies de mise à jour automatique d'un logiciel distribué (voir la sous-section 2.4.2).

2.3.8 Synthèse

Dans cette section, nous avons traité de l'adaptation d'applications à base de composants. Les critères généraux d'adaptation définis par le tableau 2.1 s'appliquent très bien aux applications à base de composants (voir les sous-sections 2.3.1 à 2.3.5).

Après avoir discuté des opérations d'adaptations d'applications à base de composants, nous avons défini les critères qui caractérisent le degré de contrôle du processus qui doit réaliser ces opérations. Ces critères illustrent principalement la mise en œuvre de l'adaptation, c'est-à-dire le « comment ». Ils vont nous permettre de comparer qualitativement les différents travaux existants sur l'adaptation d'applications à base de composants. Notons que les critères de disponibilité et de performance s'appliquent exclusivement au cas des adaptations dynamiques.

Cohérence : il fait référence à la capacité d'empêcher l'introduction d'erreurs de fonctionnement à cause du processus d'adaptation et des problèmes qu'il peut engendrer (instabilité, transfert d'état, dépendances, versions, nommage).

Performance : elle fait référence à la quantité de ressources matérielles utilisées par le processus d'adaptation. Plus cette quantité est faible, moins il y a de risque d'impact sur le logiciel adapté.

Disponibilité : elle fait référence à la capacité de garantir que certaines fonctionnalités prioritaires pourront être utilisées durant le processus d'adaptation.

Simultanéité : elle fait référence à la capacité de coordonner des adaptations qui sont déclenchées ou réalisées simultanément.

Ouverture : elle fait référence à la possibilité de configurer/adapter le processus d'adaptation avec des stratégies plus ou moins élaborées (*e.g.* adaptation paresseuse,...) d'après les principes de l'« *Open Implementation* » [Kic96].

Dans la section suivante, nous présentons plusieurs travaux concernant l'adaptation d'applications à base de composants. Nous restreignons notre étude aux adaptations qui sont dynamiques et qui portent sur l'architecture logique des applications - c'est-à-dire basées essentiellement sur l'ajout et la suppression de composants et de connexions.

2.4 Travaux sur l'adaptation dynamique d'applications à base de composants

La question de modifier dynamiquement des éléments d'une application n'est pas nouvelle. En 1976, Fabry [Fab76] a présenté une technique pour modifier à la volée des modules systèmes et des formats de structures de données. Depuis, plusieurs systèmes permettant la mise à jour dynamique ont été conçus. Les récapitulatifs les plus cités sur le sujet sont ceux de Segal & Frieder [SF93] et de Gupta & Jalote [GJ93]. Plus récemment, les travaux de Hicks [Hic01] traitent de la problématique de mise à jour dynamique.

Nous décrivons rapidement les travaux fondateurs qui concernent la problématique de l'adaptation de logiciels. Ensuite, nous présentons différents travaux mettant en œuvre la *restructuration dynamique d'application à base de composants*. Nous distinguons d'une part des travaux liés aux ADLs (« *Architecture Description Language* » [MT97]) et d'autre part les travaux liés à un modèle de composant particulier.

2.4.1 Travaux fondateurs concernant l'adaptation

En 1976, Fabry [Fab76] s'est intéressé au remplacement dynamique de l'implantation d'un type. Comme *l'implantation d'un type abstrait de données* est cachée derrière son interface, on peut agir simultanément sur toutes les instances de ce type en évitant toute incompatibilité structurelle. Pour ce faire, Fabry conserve l'ancienne implantation tant qu'elle est utilisée par certaines instances du type. Les instances qui sont passives au moment de l'adaptation pointent directement sur la nouvelle version de l'implantation. Les instances qui étaient actives pointeront vers la nouvelle version à leur prochaine exécution. Cette technique ne permet aucunement de changer la structure d'un programme.

Une approche plus générale à celle de Fabry consiste à permettre une restructuration plus diversifiée du code. Cette approche s'applique aux *systèmes à base de procédures* (basés sur des langages procéduraux comme C ou Pascal). Un des problèmes soulevé par ces modifications à grains très fin est celui de la maîtrise de leur incidence sur l'exécution du code, à cause des dépendances entre procédures. Cette approche a été utilisée dans le système PODUS [FS91].

Le modèle client-serveur est basé sur l'utilisation de modules peu flexibles à gros grain communément appelés *composants* (par exemple, les composants

EJB). Ces « composants » agissent comme un type abstrait de données, mais à un niveau plus grand. On peut modifier l'implantation des différents services fournis par le composant mais l'interface vers le serveur ne change pas [RA00]. Un défaut notable réside dans le fait que pour effectuer une petite modification dans un composant, il faut remplacer ce composant et cela perturbe d'autant plus le serveur d'applications que le composant peut être très gros.

La dynamicité des architectures apparaît dans le domaine des *ADLs* (« *Architecture Description Language* ») [MT97]. Il n'y a pas de définition communément acceptée pour un ADL, mais il s'agit de pouvoir représenter/analyser/configurer une architecture logicielle à base de composants et de connecteurs. Il existe plusieurs modèles d'ADLs (Darwin, Wright, UniCon, ...). Certains permettent de reconfigurer dynamiquement les connexions entre les éléments de l'architecture. Il est parfois possible d'ajouter, de supprimer, de remplacer dynamiquement des composants (ou des connecteurs), ou de faire évoluer les éléments existants. C'est le rôle des AMLs[8] (« *Architecture Modification Language* ») [Ore96]. On parle plus généralement d'environnements de configuration d'architectures comme Olan [Bel97] ou les frameworks « *C2-style* » [Med99].

2.4.2 Spécification d'adaptations

Plusieurs travaux utilisent des règles pour *détecter des changements* dans une application, ou pour *déclencher les opérations d'adaptations* qui s'imposent. Kaddour et Pautet [KP04] ont conçu un modèle d'adaptation où une règle s'exprime (en XML) sous la forme d'un contrat qui associe une certaine configuration du contexte d'exécution aux comportements que doit adopter l'application. Voici un exemple de règle de reconfiguration :

```
<constraints>
  <constraint resName="bandwidth" operator="greaterThan" value ="200" />
  <constraint resName="network" operator="equal" value="802.11b" />
</constraints>
<behaviours>
  <behaviour parName="serveur" value="Meteo_fr" />
  <behaviour parName="DataType" value="largeImage" />
</behaviours>
```

Ce genre de règle permet d'exprimer simplement les cas d'adaptation. La réalisation de ces adaptations peut se faire avec un simple mécanisme réactif

8. Bien souvent, l'AML et l'ADL se confondent.

qui applique un comportement donné (« *behaviours* ») dès que les contraintes de déclenchement (« *constraints* ») sont satisfaites. La règle d'adaptation ci-dessus concerne une application d'affichage de cartes météorologiques obtenues depuis un serveur. Le déclenchement de l'adaptation est conditionné par deux contraintes portant sur le réseau : (1) le réseau doit être de type WiFi (norme 802.11b) et (2) la bande passante de ce réseau doit être supérieure à 200 kbit/s. La décision associée à ce déclenchement est de choisir le serveur « Meteo_fr » et d'affecter une grande taille aux cartes météo à afficher. L'étape de réalisation n'est pas représentée directement dans la règle. La réalisation consiste simplement à prendre en compte les nouveaux paramètres d'affichage des cartes tant qu'une autre règle ne les modifie pas. On peut imaginer d'autres règles indiquant qu'il faut réduire la taille de la carte météo lorsque le réseau est peu performant. Cependant, cette approche à base de règles indépendantes n'est efficace que pour des cas simples d'adaptation.

Parfois, le processus d'adaptation doit être spécifié de manière complexe. Dans sa thèse [Ajm04], Ajmani utilise le concept de *fonctions d'ordonnancement* (« *Scheduling Functions* »). Les fonctions d'ordonnancement sont des procédures définies par un tiers (par exemple l'administrateur) pour spécifier quand une opération d'amélioration peut intervenir. Dans un système distribué, les fonctions d'ordonnancement s'exécutent sur chaque nœud, perçoivent des informations concernant un nœud et ses voisins de manière à déclencher une modification sur ce nœud ou pour stopper une réaction en chaîne de modifications. Contrairement au mécanisme précédent, la solution de Ajmani permet de réaliser automatiquement des adaptations coordonnées qui concernent plusieurs parties de l'application globale. Les fonctions d'ordonnancements permettent d'exprimer différentes stratégies de mise à jour automatique d'un logiciel distribué. Parmi les stratégies que Ajmani décrit, il y a :

- « La mise à jour gourmande » *i.e.* le plus tôt possible ;
- « La mise à jour graduelle » pour limiter le risque de pic d'activité ;
- « Mettre à jour tous les nœuds de classe C1 avant ceux de classe C2 » pour maîtriser l'ordre d'exécution des opérations de mise à jour ;
- « Mettre à jour sans créer de lieu aveugle dans la représentation géographique du système » pour vérifier par GPS que toute la partie géographique couverte par un réseau de capteurs le reste pendant et après la mise à jour (en ne mettant à jour que les nœuds redondants de ce système).

2.4.3 Adaptation dans les langages de description d'architecture

Les langages de description d'architecture (ADLs) [MT97] permettent de spécifier l'architecture d'un système logiciel en vue de son analyse. L'utilisation des ADLs se situe principalement au moment de la conception d'un système. Ils sont généralement accompagnés d'outils destinés à modéliser, analyser, simuler, générer ou encore modifier des architectures.

Dans les ADLs, l'architecture d'un système est décrite principalement en terme de *composants* (unités de calcul ou de stockage qui sont dotées d'interfaces fournies et requises), de *connecteurs* (interconnexions et règles d'interaction entre composants) et de *configurations* (graphe de composants interconnectés à travers des connecteurs).

Notons que la plupart des ADLs se limitent à des reconfigurations statiques, mais certains comme C2 [Med96], Darwin [MDK94] et Rapide [LV95] s'intéressent aux modifications dynamiques d'architectures. L'avantage de Rapide, c'est qu'il permet de spécifier avec précision le comportement dynamique d'une application (création et suppression de composants, interconnexions qui se modifient dynamiquement en fonction des propriétés des composants existants dans l'application, etc.). Toutefois, Rapide fait partie des ADLs [9] qui ne permettent pas de construire une architecture logicielle. Rapide est utilisé uniquement à des fins de vérification d'une architecture, c'est pourquoi nous ne détaillons pas ses caractéristiques. En revanche, nous décrivons C2 et Darwin.

2.4.3.1 C2

C2 est un style architectural (ensemble de contraintes compositionnelles ou comportementales guidant le développement) basé sur des composants, des connecteurs et des messages. Conçu par l'université UCI en Californie, il permet de construire des systèmes logiciels flexibles et extensibles grâce à un langage de description d'architecture C2 SADEL et un langage de modification d'architecture C2 AML [MORT96, Med96, OT98]. Cet ADL permet de définir une application sous la forme d'un réseau de composants liés par des connecteurs et s'exécutant de manière concurrente.

Description C2 permet d'exprimer des changements dynamiques dans la configuration d'une architecture puisqu'il fournit un langage pour la restructuration de l'architecture. Il permet à la fois d'opérer des restructurations im-

9. Wright [AG97] est un autre exemple connu d'ADL orienté vers la vérification formelle d'architecture.

pliquant des composants et des connecteurs. Les composants peuvent être actifs lorsqu'ils sont remplacés, mais il faut les déconnecter avant de les supprimer. Contrairement à la plupart des travaux sur le remplacement dynamique de composants, C2 supporte des changements « purement dynamiques », c'est-à-dire qu'on peut connecter le nouveau composant avant de déconnecter l'ancien. Cela suppose que le changement graduel a été déjà implanté sur les connecteurs impliqués. Par contre, C2 ne traite pas les transferts d'état.

La gestion de la cohérence lors des opérations de restructuration doivent beaucoup aux connecteurs. Un connecteur agit comme un bus de communication chargé de rediriger les messages qu'il reçoit aux composants capables de traiter ce message. Beaucoup d'informations sont spécifiées dans ces derniers : les identifiants des composants pouvant recevoir un message particulier, le nombre de composants qui le reçoivent, la façon de choisir un ou plusieurs destinataires parmi ceux ayant la capacité de traiter ce message, etc. [OT98]).

Cohérence	**Performance**	**Disponibilité**	**Simultanéité**	**Ouverture**
++	++	++	+	++

TABLE 2.2 – Critères de l'adaptation pour C2

Évaluation L'infrastructure d'adaptation utilisée dans C2 est constituée d'un ensemble de connecteurs. Ces entités matérialisent les décisions du concepteur de l'application de manière distribuée. Cette approche a l'avantage de séparer le code applicatif du code dédié aux adaptations. Cependant, la configuration des connecteurs requiert une implication humaine assez importante en terme d'implémentation si l'on souhaite spécialiser fortement les « politiques locales » d'adaptation.

Le tableau 2.2 synthétise l'évaluation de C2 par rapport à nos critères d'adaptation. Nous estimons que le critère de **cohérence** est en partie pris en compte par les connecteurs qui sont justement chargés de gérer de manière flexible et automatique les ajouts et suppressions dynamiques de composants. Néanmoins, l'implémentation de ce réseau de « gestionnaires d'adaptation » repose sur le développeur de l'application. De plus, les auteurs de C2 font l'hypothèse que les composants n'ont pas d'état, ce qui simplifie le problème du remplacement dynamique de composants. Le besoin de **performance** pendant l'adaptation n'est pas spécifiquement pris en compte dans C2. Toutefois, l'impact des ajouts et suppressions de composants semble négligeable compte-tenu que seuls les opérations d'adaptation simples (ajouts ou suppressions de composants) sont disponibles. Le besoin de **disponibilité** des services offerts par l'application pendant l'adaptation n'est pas spécifiquement pris en compte, mais il peut être traité par des connecteurs, selon

l'implémentation proposé par le développeur. Le cas des adaptations **simultanées** n'est pas pris en compte car les adaptations sont toujours à l'initiative d'un humain (administrateur de l'application). L'infrastructure d'adaptation fonctionne selon un modèle mono-tâche, ce qui signifie que lorsqu'il faut réaliser plusieurs adaptations, elles sont exécutées l'une après l'autre. Il paraît très difficile de coordonner des adaptations complexes, étant donné que chaque connecteur peut avoir sa propre stratégie pour gérer - localement - les adaptations. Enfin, nous considérons que le degré d'**ouverture** de l'infrastructure d'adaptation de C2 est moyen car il faut redévelopper une bonne partie des connecteurs lorsqu'on souhaite personnaliser la gestion distribuée des adaptations.

2.4.3.2 Darwin

Darwin est un ADL développé au Collège Impérial de Londres [MDK94]. Il fait suite aux travaux sur un précédent langage pour la configuration et la reconfiguration dynamique d'applications distribuées basées sur le concept de module (CONIC [MS89]). Il utilise un modèle de composants hiérarchiques et sans connecteurs. Sa particularité est de permettre l'instantiation automatique de schémas d'interactions complexes entre les composants, avant, après ou en cours d'exécution. Les interactions complexes sont spécifiées en particulier à l'aide de constructions syntaxiques telles que la condition `when` et l'itérateur `forall`.

Description Dans Darwin, un composant est une unité de traitement qui propose des interfaces fournies et des interfaces requises. Ces interfaces ne désignent que le type de communication utilisé ou autorisé à venir appeler une fonction du composant. Il existe deux types de composants dans Darwin : les composants primitifs représentent un processus et les composants composites représentent les entités de configuration puisqu'ils contiennent la description des interconnexions internes au composite. Une application est un composite.

Darwin permet d'exprimer des changements dynamiques d'architectures grâce à un mécanisme d'instanciation dynamique. Une instantiation dynamique de composants peut être paresseuse ou directe. L'*instantiation dynamique paresseuse* permet de reporter/retarder l'instantiation de composants décrits dans l'architecture grâce à des règles prévues par le concepteur de l'architecture. Par contre, l'*instantiation dynamique directe* permet à l'administrateur d'une architecture de faire des changements arbitraires et non anticipés. Les modifications d'architecture engendrées par instantiation dynamique directe ne sont pas capturées dans la description d'architecture et sont ainsi complètement incontrôlables.

Nous pouvons affirmer que la problématique du réassemblage de composants n'est que moyennement prise en compte dans Darwin car (1) les composants instanciés dynamiquement et automatiquement ne peuvent plus être supprimés ou déconnectés et (2) les modifications non anticipées de l'architecture ne sont pas maîtrisables.

Cohérence	**Performance**	**Disponibilité**	**Simultanéité**	**Ouverture**
++	+	++	++	++

TABLE 2.3 – Critères de l'adaptation pour Darwin

Évaluation L'infrastructure d'adaptation utilisée dans Darwin est constituée du langage de spécification des configurations et d'un gestionnaire de configurations.

Le tableau 2.3 synthétise l'évaluation de Darwin par rapport à nos critères d'adaptation. Nous estimons que le critère de **cohérence** est en partie pris en compte par l'utilisation du concept de « quiescence » [KM90, BAS+04]. Il s'agit de s'assurer que les composants impliqués dans une opération d'adaptation sont dans un état quiescent, c'est-à-dire au repos et prêts pour les modifications. Pour ce faire, il faut suspendre temporairement les appels entrants (en les stockant pour une réutilisation ultérieure à l'opération d'adaptation) autour des composants et attendre un certain délai. La problématique du transfert d'état n'est pas spécifiquement abordée. Le besoin de **performance** pendant l'adaptation n'est pas du tout pris en compte dans Darwin. Le besoin de **disponibilité** des services offerts par l'application pendant l'adaptation n'est pas spécifiquement pris en compte, mais il est possible d'instancier dynamiquement un composant analogue au composant qui est en train d'être adapté, afin que les services offerts restent disponibles en permanence. Néanmoins, c'est au concepteur de l'application de prendre en charge les éventuelles incohérences dues à l'utilisation des services d'un composant potentiellement obsolète. Le cas des adaptations **simultanées** se limite à instantier ou supprimer séquentiellement des composants ou des connexions, grâce l'opérateur d'itération du langage Darwin. Enfin, nous considérons que le degré d'**ouverture** de l'infrastructure d'adaptation de Darwin est moyen car son ouverture est basée sur la notion de composant composite : c'est explicite mais cela manque d'abstraction.

2.4.4 Adaptation dans les modèles de composants

Il existe de nombreux travaux qui traitent de l'adaptation d'applications à base de composants, notamment en fournissant une infrastructure d'adap-

tation particulière. Certains de ces travaux utilisent des modèles de composants industriels tels EJB [10] ou Fractal [11], tandis que d'autres proposent leurs propres modèles de composants.

2.4.4.1 SAFRAN

SAFRAN [Dav05, DL05, DL03] est une extension du modèle de composant Fractal visant à supporter le développement des composants auto-adaptatables. Développé à l'École des Mines de Nantes, il s'applique notamment au domaine de l'informatique autonomique dans lequel les applications doivent s'adapter fréquemment.

Description SAFRAN est basé sur l'introduction d'une extension réflexive permettant de modifier de manière transparente le comportement d'un composant en fonction de son contexte d'exécution. Il utilise également la programmation par aspects pour développer l'adaptation en tant qu'aspect à tisser dynamiquement dans les applications. La programmation des aspects sous forme de politique réactive est réalisée grâce à un langage dédié (FScript).

Dans SAFRAN, les politiques d'adaptation sont basées sur le modèle ECA (Évènement-Condition-Action). Les actions permettent de réassembler une application, c'est-à-dire un composant composite. Le déclenchement automatique de ces actions consiste à tester les évènements issus du contexte. La modélisation de ce contexte est basée sur un gestionnaire générique de contexte (WildCAT).

L'adaptation de chaque composant est pilotée à l'aide d'une politique réactive individuelle basée sur des règles ECA. Cette approche peut conduire à des incohérences puisque chaque composant possède son propre ensemble de règles et s'adapte indépendamment des autres composants. De telles incohérences de l'application sont détectées a posteriori. Les adaptations en cours sont annulées et l'application est ramenée à son état précédant l'adaptation. Le coût de ces opérations imprévues peut être prohibitif, notamment dans le cas de systèmes contraints en ressources.

Cohérence	Performance	Disponibilité	Simultanéité	Ouverture
+++	+	++	++	+++

Table 2.4 – Critères de l'adaptation pour SAFRAN

10. « *Enterprise Java Bean* »
11. `http://fractal.objectweb.org`

Évaluation L'infrastructure d'adaptation utilisée dans SAFRAN est assez sophistiquée car elle inclut non seulement un modèle de politiques d'adaptation mais aussi un gestionnaire générique de contexte et un langage de reconfiguration. Ainsi, un concepteur d'applications dispose de tous les outils nécessaires pour définir facilement une application auto-adaptable.

Le tableau 2.4 synthétise l'évaluation de SAFRAN par rapport à nos critères d'adaptation. Nous estimons que le critère de **cohérence** est complètement pris en compte par le mécanisme qui vérifie a posteriori que les adaptations ne provoquent pas d'incohérence, quitte à les annuler le cas échéant, ainsi que le transfert d'état. Le besoin de **performance** pendant l'adaptation n'est pas assez pris en compte car la vérification a posteriori des adaptations est systématique. Ainsi, toutes les adaptations qui n'aboutissent pas sont détectées et annulées. Mais, ce processus a des impacts négatifs sur la durée totale des opérations d'adaptation et sur la consommation de ressources matérielles par l'infrastructure d'adaptation. Le besoin de **disponibilité** des services offerts par l'application pendant l'adaptation n'est pas spécifiquement pris en compte, mais il est en partie satisfait grâce au mécanisme de transfert d'état et de temporisation des appels vers le composant parent du composant concerné par l'adaptation. Cependant, le nombre de composants bloqués inutilement peut être grand. Le cas des adaptations **simultanées** est partiellement pris en compte car d'après la nature hiérarchique du modèle de composants, lorsqu'il faut réaliser plusieurs opérations d'adaptations, elles doivent être coordonnées par le composant le plus bas dans la hiérarchie qui contiennent tous les composants impliqués (*i.e.* « le parent le plus proche »). Les adaptations ne peuvent pas s'exécuter en parallèle, par contre, le nombre de composants à stopper lors d'une adaptation complexe est arbitrairement grand, ce qui est un frein au caractère dynamique de l'adaptation. Enfin, nous considérons que le degré d'**ouverture** de l'infrastructure d'adaptation de SAFRAN est important car elle est facilement configurable à travers une politique d'adaptation, le langage de reconfiguration ainsi que le gestionnaire de contexte qui est extensible.

2.4.4.2 Think

Think est un environnement de développement de noyaux de systèmes d'exploitation à base de composants [Fas01, Sen03, SCS02]. Développé au laboratoire LSR de l'INRIA Rhone-Alpes, il est utilisé pour créer différents types d'architectures logicielles, selon les infrastructures cibles et les domaines d'application. Une version enrichie de Think (parfois appelée eThink) a été dotée d'un modèle de composition hiérarchique et d'un modèle de reconfiguration, pour autoriser la définition de composants munis de capacités d'auto

description (introspection) et de reconfiguration dynamique.

Description Think fournit un vaste choix d'opérations d'adaptation (ajout de composants, suppression de liaisons, ...). Le modèle de composants sous-jacent est Fractal [12], un modèle de composants hiérarchiques ayant des interfaces de contrôle. Un des principaux apports de Think tient dans le fait que l'utilisation des mécanismes d'adaptation est optionnelle. Cette propriété est particulièrement intéressante pour développer des systèmes embarqués (fortement contraints), car elle garantit que le surcoût engendré en temps d'exécution par ces mécanismes ne survient que lorsque c'est nécessaire, c'est-à-dire en période d'adaptation.

Les opérations de restructuration dans Think concernent un ou plusieurs composants, mais elle peuvent aussi porter sur des liaisons entre composants. Think utilise le transfert d'état aussi bien pour ses attributs que pour l'état de ses activités (processus en cours). Mais, dès que les deux composants n'ont pas la même représentation interne, la charge du programmeur s'accroît afin de veiller à la cohérence du système.

Par ailleurs, il n'y a pas de procédure générale pour la gestion des dépendances entre composants. Les mécanismes permettant d'obtenir et de modifier les dépendances entre composants sont présents lorsqu'on ajoute ou supprime des composants ou des liaisons. Mais, les vérifications qui sont associées à ses changements ne sont que suggérées. C'est au programmeur de gérer les dépendances lorsqu'il le juge nécessaire. Cette liberté permet de ne pas alourdir inutilement l'infrastructure et les composants.

Cohérence	Performance	Disponibilité	Simultanéité	Ouverture
++	++	++	++	++

TABLE 2.5 – Critères de l'adaptation pour Think

Évaluation L'infrastructure d'adaptation utilisée dans Think est minimaliste mais extensible. Il définit un langage de description d'architecture permettant de spécifier l'application à base de composants. Think permet au développeur de définir un certain nombre de contraintes d'intégrité au cours de la vie d'un système pour assurer la cohérence de ses reconfigurations. Par exemple, (1) les *contraintes de cohérence structurelle* concernent essentiellement la conformité de type pour connecter une interface fournie et une interface requise, (2) les *invariants de comportement* peuvent garantir que des adaptations simultanées ne se perturbent pas mutuellement et (3) les

12. `http://fractal.objectweb.org`

contraintes sémantiques sont censées vérifier qu'une signature donnée correspond bien à un comportement prévu. Il est clair que le poids de la liberté qui est donnée pour réaliser des opérations d'adaptation sûres pèse sur le développeur.

Le tableau 2.5 synthétise l'évaluation de Think par rapport à nos critères d'adaptation. Nous estimons que le critère de **cohérence** est en partie pris en compte dans Think par différents mécanismes tels que le transfert d'état, le transfert d'activités en cours et l'utilisation de contraintes d'intégrité. Malheureusement, ces mécanismes ne sont que suggérés. Le besoin de **performance** pendant l'adaptation est moyennement pris en compte car il faudrait implémenter des contraintes d'intégrité spécifiques. Néanmoins, leurs auteurs mettent plutôt l'accent sur la performance de l'application lorsqu'il n'y a pas d'adaptations en cours. Le besoin de **disponibilité** des services offerts par l'application pendant l'adaptation n'est pas spécifiquement pris en compte, mais il est en partie satisfait grâce au mécanisme de transfert d'état, de transfert d'activités lors du remplacement dynamique des composants et de temporisation des appels vers le composant parent du composant concernée par l'adaptation. Cependant, le nombre de composants bloqués inutilement peut être grand. Le cas des adaptations **simultanées** est partiellement pris en compte car d'après la nature hiérarchique du modèle de composants, lorsqu'il faut réaliser plusieurs opérations d'adaptations, elles doivent être coordonnées par le composant le plus bas dans la hiérarchie qui contient tous les composants impliqués. Enfin, nous considérons que le degré d'**ouverture** de l'infrastructure d'adaptation de Think est moyen car elle est configurable surtout pour les développeurs expérimentés.

2.4.4.3 SOFA

SOFA (« *SOFtware Appliance* ») est une plateforme développée à l'Université Charles en République Tchèque pour définir des applications distribuées à base de composants [PBJ98, BHP06]. Dans SOFA, une application est vue comme une hiérarchie de composants. Un de ses objectifs est de permettre la personnalisation de composants en cas de besoin et de réviser leurs interconnexions avec les autres parties d'une telle application grâce à des connecteurs. Par exemple, il permet de remplacer une liaison locale par une liaison distante entre deux composants. Pour ce faire, SOFA utilise des connecteurs qui sont adaptables à la manière de certains travaux sur les architectures logicielles (par exemple, le modèle C2).

Description SOFA traite les problèmes de transferts d'état, de versions [Bra03], de nommages [HT03], de dépendances entre compo-

sants [PBJ98]. La mise à jour dynamique des composants est fondée sur DCUP (« *Dynamic Component UPgrading* »), une extension du modèle de composants de SOFA.

Les composants disposent d'une partie permanente qui est responsable de leur cycle de vie (notamment des mises à jour) et d'une partie remplaçable (code fonctionnel ou sous-composants). Mais, pour permettre ces remplacements (partiels) de composants, les développeurs doivent implémenter systématiquement certaines interfaces. En outre, la nouvelle version d'un composant doit fournir les mêmes interfaces que l'ancienne version. La tâche du développeur de composants est d'autant plus complexe quand il s'agit de réaliser des composants dynamiquement adaptables et de gérer les problèmes de cohérence qui en découlent. La spécification d'une partie remplaçable au niveau du modèle de composant, facilite les opérations d'adaptation. Pour assurer la stabilité du système pendant un remplacement, les composants sont stoppés à l'instant précis du remplacement.

Récemment, SOFA 2.0 [BHP06] a été proposé pour pouvoir mieux contrôler les adaptations architecturales d'applications, en particulier grâce à trois patrons de reconfiguration. Le patron « *nested factory* » permet d'utiliser un composant particulier pour créer d'autres composants, le patron « *component removal* » permet de supprimer des composants créés dynamiquement et le patron « *utility interface* » permet de connecter des composants quelque soient leurs positions dans la hiérarchie de l'application. Ces patrons gèrent respectivement l'ajout de composants, la suppression de composants et les connexions entre composants quelles que soient leurs positions dans la hiérarchie de l'application. Enfin, la partie non remplaçable des composants est désormais reconfigurable et extensible avec une approche basée sur des micro-composants. SOFA offre donc une séparation nette entre la préoccupation applicative et la préoccupation administrative d'une application à base de composants.

Cohérence	**Performance**	**Disponibilité**	**Simultanéité**	**Ouverture**
++	+	++	++	++

TABLE 2.6 – Critères de l'adaptation pour SOFA

Évaluation L'infrastructure d'adaptation utilisée dans SOFA fait intervenir tout un ensemble d'entités logicielles (connecteurs, contrôleur interne de composants, fabriques de composants, ...). Ces entités matérialisent les décisions du concepteur de l'application de manière distribuée. Néanmoins, l'approche manque d'abstraction. Son mode très « opératoire » s'adresse sur-

tout à des développeurs expérimentés car l'infrastructure d'adaptation doit être programmée pour chaque application.

Le tableau 2.6 synthétise l'évaluation de Think par rapport à nos critères d'adaptation. Nous estimons que le critère de **cohérence** est en partie pris en compte dans SOFA car DCUP ainsi que les patrons « *nested factory* », « *component removal* » et « *utility interface* » gèrent de manière rigoureuse les différentes opérations de restructuration de base. Cependant, la logique d'adaptation d'une application doit être répartie dans un ensemble d'entités logicielles qu'il faut programmer. A priori, il n'y aucune garantie que toutes ces entités se coordonnent de manière cohérente. Le besoin de **performance** pendant l'adaptation n'est pas spécifiquement pris en compte dans SOFA. Le besoin de **disponibilité** des services offerts par l'application pendant l'adaptation n'est pas spécifiquement pris en compte, mais il est en partie satisfait grâce à DCUP qui ne stoppe qu'une partie de l'application. Le cas des adaptations **simultanées** est partiellement pris en compte car l'infrastructure d'adaptation fait intervenir un ensemble d'entités logicielles spécialisées et réparties dans l'application. Les auteurs ne précisent pas si des adaptations simultanées peuvent être coordonnées selon une stratégie précise. Nous supposons que si une adaptation concerne un ensemble de composants, les étapes de cette adaptation complexe seront déterminées selon les choix des patrons de reconfiguration qui sont définis avant tout pour garantir la cohérence de l'application. Enfin, nous considérons que le degré d'**ouverture** de l'infrastructure d'adaptation de SOFA est moyen car elle est configurable surtout pour les développeurs expérimentés.

2.4.4.4 CASA

CASA (« *Contract-based Adaptive Software Architecture* ») fournit un framework permettant l'adaptation dynamique d'applications [MG05, MG03]. Il est développé à l'université de Zurich afin d'adapter dynamiquement une application en réponse à des changements dans son environnement d'exécution.

Description Deux types de changements sont considérés pour déclencher des adaptations : (1) les changements concernant des informations contextuelles (localisation de l'utilisateur, objets environnants, température ambiante, ...) qui peuvent influencer le service fourni par une application, et (2) les changements qui concernent la disponibilité des ressources (bande passante, autonomie des batteries, connectivité, ...) qui sont utiles à l'application qui fournit ce service. Dans le premier cas, on souhaite maximiser la pertinence du comportement de l'application pour le service qu'elle doit

fournir, tandis que dans le second cas, on préfère minimiser la consommation de ressources de ce service par l'application.

Pour réaliser l'adaptation d'une application, CASA met en œuvre différents mécanismes. Chaque mécanisme est dédié à une cible d'adaptation particulière : les changements dynamiques dans les services de bas niveau (transmission de données et compression) utilisent des techniques réflexives, le tissage et le détissage dynamique d'aspects pour les adaptions transversales (comme changer le comportement de sécurité et de persistance) sont basés sur le système PROSE [PGA02], les changements dynamiques dans les attributs de l'application s'appuient sur des méthodes de « *callback* », et la recomposition dynamique de composants est dédiée aux adaptations qui impliquent l'ajout, la suppression et le remplacement de composants. Dans CASA, chaque adaptation peut impliquer plusieurs de ces mécanismes, selon la politique d'adaptation.

Les auteurs de CASA travaillent avec leur propre modèle de composants et proposent un mécanisme original pour le remplacement dynamique de composants. Dans CASA, un composant est l'instance d'une classe. Un composant peut être remplacé dynamiquement par une autre instance de sa classe ou d'une classe alternative. Chaque ensemble de classes interchangeables est associé à une unique classe « *Handle* ». Les états des instances des classes interchangeables sont transférés entre elles en passant systématiquement par des instances de la classe « *Handle* » associée. Ce mécanisme permet que les remplacements dynamiques soient transparents vis-à-vis des composants.

La cohérence des adaptations dynamiques est basée sur la stabilisation des composants impliqués. Pour atteindre un état stable avant le remplacement d'un composant C, la tâche du développeur n'est pas aisée, car il faut non seulement désactiver C pour l'isoler des appels externes mais surtout suspendre l'exécution interne de C. Il n'est pas possible de garantir dans tous les cas que le composant remplaçant pourra reprendre correctement l'exécution du composant remplacé. C'est pourquoi le modèle de CASA permet d'utiliser deux sortes de stratégies de remplacements dynamiques de composants : l'une est dite paresseuse (car elle attend l'arrêt complet du composant avant de le remplacer), et l'autre est dite avide (car elle force le composant à s'arrêter quitte à reprendre son exécution après le remplacement). Le développeur doit implémenter la stratégie la plus appropriée au niveau de chaque classe « *Handle* ».

Évaluation L'infrastructure d'adaptation de CASA a l'avantage d'utiliser des spécifications de haut niveau comme une politique d'adaptation ou des contrats. Une politique d'adaptation associe directement un contexte à

Cohérence	**Performance**	**Disponibilité**	**Simultanéité**	**Ouverture**
+++	++	+++	+++	+++

TABLE 2.7 – Critères de l'adaptation pour CASA

un ensemble de configurations alternatives ayant des besoins différents. Les contrats décrivent les besoins de l'application et sont utilisés par la politique d'adaptation pour contrôler le déroulement des adaptations. Ainsi, les adaptations sont appliquées lorsque c'est nécessaire et de manière complètement automatique. Enfin, les auteurs disent que la politique d'adaptation peut également être modifiée dynamiquement car elle est dans un fichier XML séparé de l'application.

Le tableau 2.7 synthétise l'évaluation de CASA par rapport à nos critères d'adaptation. Nous estimons que le critère de **cohérence** est complètement pris en compte dans CASA par l'implémentation d'une stratégie paresseuse au niveau de chaque classe « *Handle* » qui le nécessite. De plus, les assemblages résultant des adaptations correspondent à des configurations définies par le concepteur de l'application et donc supposées sûres. Le besoin de **performance** pendant l'adaptation n'est pas spécifiquement pris en compte, mais il est en partie satisfait car le concepteur a la possibilité d'utiliser une stratégie de remplacement avide pour mettre l'accent sur la rapidité de l'adaptation ou une stratégie paresseuse pour perturber le moins possible l'application. Toutefois, la stratégie de remplacement avide n'est plus performante que la stratégie paresseuse que lorsqu'il y a assez de ressources matérielles (notamment CPU) disponibles pour l'infrastructure d'adaptation. Le besoin de **disponibilité** des services offerts par l'application pendant l'adaptation est pris en compte car grâce à la stratégie paresseuse, il est possible de favoriser la disponibilité de services du composant à remplacer, quitte à retarder quelque peu cette adaptation. Le cas des adaptations **simultanées** est complètement pris en compte car une opération d'adaptation consiste à adopter une des configuration prévues dans la politique d'adaptation et peut donc concerner à la fois différents niveaux (services de base, services transversaux, services applicatifs) et différents composants logiciels. Enfin, nous considérons que le degré d'**ouverture** de l'infrastructure d'adaptation de CASA est important car non seulement elle est facilement configurable à travers une politique d'adaptation mais surtout le concepteur de l'application peut appliquer différentes stratégies de mise en œuvre de l'adaptation.

2.5 Bilan de l'étude de l'adaptation des applications à base de composants

Différentes raisons peuvent justifier le besoin d'adaptation en génie logiciel et notamment, en ce qui concerne les applications à base de composants (maintenance, évolution des fonctionnalités ou du contexte, etc.). Cependant, nous avons constaté dans notre étude comparative que les adaptations dynamiques sont difficiles à mettre en œuvre, surtout lorsqu'on est exigeant sur la qualité de ces adaptations.

Outre les problèmes de maintien de cohérence, la restructuration dynamique pose différents problèmes difficiles. Ainsi, des efforts conséquents restent à faire pour améliorer certains aspects des adaptations : la réduction des coûts des adaptations et la maximisation de la disponibilité de certaines fonctionnalités d'une application lors d'une adaptation, la coordination automatique des adaptations qui se chevauchent sur le plan temporel ou le plan spatial.

Le tableau 2.8 récapitule les résultats de notre étude sur l'adaptation dynamique des applications à base de composants. Aucun des travaux étudiés ne satisfait totalement l'ensemble de nos critères de qualité. Néanmoins, CASA et SAFRAN prennent en compte la plupart des critères car ils proposent une infrastructure d'adaptation très ouverte et basée sur des concepts de haut niveau, telle que les politiques d'adaptation. Think et SOFA sont eux modérément ouverts et requièrent une participation humaine plus importante, car la personnalisation des infrastructures d'adaptation qu'ils proposent passe par une phase de développement qui peut être source d'erreurs d'implémentation. Enfin, C2 et Darwin fournissent des ADLs qui prennent en charge des restructurations dynamiques, mais les opérations d'adaptation qu'elles mettent en œuvre sont souvent assez basiques comparées aux nombreuses adaptations qui sont a priori envisageables dans un assemblage de composants.

	Cohérence	**Performance**	**Disponibilité**	**Simultanéité**	**Ouverture**
Travaux dans le domaine des langages de description d'architecture					
C2	++	+	++	++	++
Darwin	++	+	++	++	++
Travaux liés à un modèle de composants					
SAFRAN	+++	+	++	++	+++
Think	++	++	++	++	++
SOFA	++	+	++	++	++
CASA	+++	++	+++	+++	+++

TABLE 2.8 – Comparatif des ABCs adaptables

En guise de bilan de ce chapitre d'état de l'art, nous pouvons dire qu'il

est utopique d'avoir une infrastructure d'adaptation dynamique qui réalise automatiquement des restructurations parfaites en toutes situations. Il est essentiel de pouvoir paramétrer l'infrastructure d'adaptation en fonction de certains choix stratégiques (*e.g.* favoriser la disponibilité de certains services applicatifs) ou de contraintes techniques (*e.g.* manque de ressources matérielles). Cela permet que le concepteur d'une application puisse contrôler les adaptations et donc exercer pleinement le rôle de gestionnaire de l'application. De plus, pour pouvoir utiliser l'infrastructure d'adaptation de la manière la plus aisée qui soit, il est souhaitable que le fonctionnement de l'infrastructure d'adaptation soit explicite et naturel du point de vue d'un concepteur d'application. C'est pourquoi, une infrastructure d'adaptation doit idéalement utiliser des concepts de haut niveau comme des contrats, des configurations et des politiques d'adaptation.

On ne peut que regretter le fort couplage qui existe généralement entre une infrastructure d'adaptation et le modèle de composants utilisé. De plus, les spécifications des adaptations sont souvent fortement couplées à une application particulière. L'objectif majeur de la programmation par composants étant la *réutilisabilité*, il serait souhaitable d'avoir une approche générique pour la construction et l'adaptation d'applications à base de composants, à travers une infrastructure qui ne ferait que très peu d'hypothèses sur les composants à utiliser. En effet, cela permettrait d'avoir une infrastructure avec laquelle :

1. on peut utiliser la plupart des modèles de composants répandus,
2. on peut facilement réutiliser des spécifications d'une application à base de composants avec d'autres composants que ceux prévus initialement, et
3. on peut simplement réutiliser des spécifications d'adaptations dans d'autres applications que celles prévues initialement.

Chapitre 3

Adaptabilité d'agents à base de composants

*Les **Systèmes Multi-Agents** (SMAs) sont des systèmes composés d'entités autonomes, appelées* agents*, pouvant interagir entre elles.* Il n'y a pas une définition unanimement acceptée pour la notion d'agent, mais plusieurs définitions sont régulièrement citées, notamment les deux suivantes.
Selon Jennings & Wooldridge,

> « un **agent** est un système informatique situé dans un environnement, et qui agit d'une façon autonome pour atteindre les objectifs pour lesquels il a été conçu » [JW98].

Selon Ferber,

> « un **agent** est une entité autonome, réelle ou abstraite, qui est capable d'agir sur elle-même et sur son environnement, qui, dans un univers multi-agent, peut communiquer avec d'autres agents, et dont le comportement est la conséquence de ses observations, de ses connaissances et des interactions avec les autres agents » [Fer95].

Les usages des SMAs sont nombreux et variés car ils permettent de « coordonner des opérations distribuées, de simuler de manière décentralisée des systèmes ouverts toujours plus complexes, d'accompagner le développement de l'infrastructure de communication, de modéliser et de réaliser des systèmes à un niveau d'abstraction toujours plus élevé » [OFTA02]. Du fait de la diversité des besoins, il existe une grande variété de modèles d'agents et d'architectures d'agents.

> « Un **modèle d'agents** définit le cadre dans lequel le concepteur pourra exprimer les comportements de l'agent. [. . .] Il doit ainsi répondre aux interrogations suivantes : (i) Comment traiter un

> message ? (ii) Comment associer le traitement d'un message au programme réalisant ce traitement ? (iii) Comment décider de la prochaine action à effectuer (du prochain message à émettre) ? » [Sec03].

En principe, les propriétés d'un modèle d'agents dépendent du domaine d'application visé par son concepteur. Néanmoins, la littérature montre qu'on peut classer les différents modèles d'agents selon différentes typologies [BMP06, Boi01, Mül97, WJ95]. Chaque type de modèles constitue ce qu'on appelle une architecture d'agents et les modèles n'ayant pas de type explicite sont appelés des modèles *ad hoc*.

> « Une **architecture d'agents** est la structure logicielle (ou matérielle) qui, à partir d'un ensemble d'entrées, produit un ensemble d'actions sur l'environnement ou sur les autres agents. Sa description est constituée des composants(correspondant aux fonctions) de l'agent et des interactions entre ceux-ci (flux de données et de contrôle) » [Boi01].

Nous considérons qu'en adhérant à une architecture d'agents particulière, un concepteur spécialise son modèle d'agents pour avoir - dans une certaine mesure - des avantages et des compromis prévisibles.

Dans ce chapitre, nous nous intéressons à l'adaptabilité des agents et plus particulièrement aux modèles d'agents capables de changer leur architecture. L'*adaptabilité* d'un système réside en la possibilité de le modifier pour se conformer à des situations nouvelles (voir la section 2.2). Dans beaucoup de modèles d'agents, le degré d'adaptabilité des agents est plutôt faible, voire nul [AC87, Bro90]. Cependant, dans le cas des Systèmes Multi-Agents ouverts, un haut degré d'adaptabilité des agents est requis pour prendre en considération des changements contextuels imprévisibles. *Par **SMA ouvert**, nous désignons un SMA qui admet certaines modifications après la phase de conception.* Ces modifications peuvent survenir au *niveau système*, en ajoutant ou en supprimant des agents, ou au *niveau agent*, en modifiant des agents. Notons qu'une adaptation d'agent peut être requise après l'ajout ou la suppression d'autres agents. Par exemple, imaginons qu'un agent médiateur ait pour rôle de superviser et coordonner les actions d'un groupe d'agents d'exploration. Si l'agent médiateur est supprimé (ou durablement indisponible), alors le fonctionnement global du SMA sera affecté voire stoppé. Les agents restants devront s'adapter d'une manière ou d'une autre (par exemple, trouver un nouvel agent médiateur ou adopter une nouvelle façon d'interagir). Comme les changements contextuels peuvent être fréquents ou imprévisibles dans un SMA ouvert, l'adaptation doit être un concept central du modèle d'agents. Donc, l'adaptabilité des agents devrait apparaître explicitement

dans leur architecture. Par conséquent, le problème auquel les concepteurs sont confrontés est de prendre en charge (*i.e.* faire des choix de conception tendant à minimiser) la perturbation d'un système dû à des changements imprévus et répétés, sans pour autant nuire à l'autonomie des agents qui composent ce système.

Ce chapitre est structurée comme suit :

- dans la section 3.1, nous explicitons la notion d'architecture d'agents et son utilisation pour concevoir et adapter des agents,
- dans la section 3.2, nous faisons une étude comparative de différents modèles d'agents auto-adaptables sans composants,
- dans la section 3.3, nous faisons une étude comparative de différents modèles d'agents auto-adaptables à base de composants,
- dans la section 3.4, nous dressons une synthèse des travaux existants sur les agents auto-adaptables, en insistant plus particulièrement sur les manques.

3.1 Architectures d'agents

Les architectures d'agents sont très variées et il en existe plusieurs typologies [BMP06, Boi01, Mül97, WJ95]. Dans cette section, nous explicitons les avantages des principaux types d'architectures d'agents, ainsi que leurs inconvénients. Puis, nous nous intéressons au cas particulier des agents à base de composants. Enfin, nous nous focaliserons sur la problématique de l'adaptabilité des agents.

3.1.1 Différents types d'architecture d'agents

Il existe plusieurs typologies d'architecture d'agents. Chacune d'elle permet d'établir une classification en mettant l'accent sur un aspect particulier : mode de raisonnement, mode de coordination, etc.

3.1.1.1 Selon les capacités de raisonnement

L'une des classifications les plus prédominantes dans la littérature ([Boi01, Mül97, WJ95]) se base sur le type de raisonnement mis en œuvre dans l'agent, c'est-à-dire leur degré de cognition. Les principaux types considérés sont les *architectures réactives*, les *architectures délibératives* et les *architectures hybrides*.

- Une **architecture réactive** se caractérise par le fait que l'agent « décide » des actions à faire à partir de peu d'informations. Un tel agent

a une représentation très basique du monde, voire aucune représentation symbolique de l'environnement. Ainsi, ses actions sont directement déterminées par les percepts de l'instant présent. Parmi les exemples connus d'architectures réactives, on trouve l'architecture de subsomption de Brooks [Bro90] ou encore le célèbre PENGI de Agre & Chapman [AC87]. Les principaux avantages de cette architecture sont sa robustesse et sa simplicité. Par contre, elle n'est pas appropriée pour définir des agents au comportement complexe ou optimal dans la plupart des cas.

- Une **architecture délibérative** [1] se caractérise par le fait que l'agent décide des actions à faire selon un mécanisme complexe appelé *délibération*. Pour ce faire, l'agent doit avoir une représentation symbolique du monde basée sur des concepts logiques tels que les croyances, les buts, ou encore les intentions. Cette architecture s'appuie notamment sur les travaux liés à la planification en Intelligence Artificielle comme STRIPS [FN71]. L'exemple le plus connu d'architectures délibératives est l'architecture de type BDI (« *Belief-Desire-Intention* ») de Rao & Georgeff dans laquelle un agent est décrit par (i) ses Croyances, qui déterminent sa connaissance courante du monde, (ii) ses Désirs, qui déterminent ses buts, et (iii) ses Intentions, qui sont générées à partir des Croyances et des Désirs pour déterminer les meilleures actions possibles [RG91]. L'approche BDI a notamment été mise en œuvre par Bratman et al. dans l'architecture IRMA [BIP88] et par Nwana et al. dans la plateforme ZEUS [NNL98]. Grâce aux architectures délibératives, il est possible de définir des agents au comportement complexe et visant un résultat optimal. Malheureusement, ce genre de comportement est coûteux en temps d'exécution et n'est pas robuste car il faut faire face à des problèmes logiques qui sont *indécidables*.
- Une **architecture hybride** résulte d'une combinaison des deux précédents types d'architectures, afin de bénéficier des avantages des deux approches. Elle se caractérise par des agents dont le comportement n'est ni complètement réactif, ni complètement délibératif. Ce type d'architecture se traduit généralement par des agents structurés en différents modules interconnectés, chaque module pouvant être conçu selon une perspective d'agent réactif ou d'agent délibératif. Parmi les exemples connus d'architectures hybrides, on trouve notamment l'architecture PRS de Georgeff & Lansky [GL87] qui utilise une approche de type BDI mais dont les Désirs sont des comportements qui peuvent être invoqués de manière cognitive ou de manière réactive lors de certains

1. Les architectures délibératives sont parfois appelées architectures cognitives.

changements dans l'environnement. Par ailleurs, il y a les architectures organisées en couches comme l'architecture TouringMachine développée par Ferguson dans sa thèse [Fer92a] et l'architecture InteRRaP de Müller et al. [Mül96].

3.1.1.2 Selon les capacités de coordination

Une autre typologie possible pour les architectures d'agents est celle de Boissier [Boi01]. Il distingue trois types d'architectures d'agents, caractérisées par leur degré de coordination : les *architectures d'agents autonomes*, les *architectures d'agents interagissants* et les *architectures d'agents sociaux*. Ce découpage fait aussi référence à l'approche VOYELLES [Dem01], dans laquelle un SMA est modélisé selon quatre dimensions : Agent (A), Environnement (E), Interaction (I) et Organisation (O). Chacune des trois architectures d'agents précitées se caractérise par la présence de certaines capacités qu'on associe aux dimensions de VOYELLES : comportement autonome (A), perception et action sur l'environnement (E), représentation des autres agents et communication entre agents (I), coopération entre agents (O).

- Une **architecture d'agent autonome** se caractérise par le fait que l'agent possède des capacités d'action et de perception sur son environnement. Cette architecture est ainsi constituée des fonctions exprimant les facettes A et E. L'agent possédant une telle architecture n'a pas explicitement [2] une capacité de coopération avec d'autres agents. Pengi [AC87], TouringMachine [Fer92a] ou encore IRMA [BIP88] sont des exemples d'architectures d'agents autonomes.
- Une **architecture d'agent interagissant** se caractérise par le fait que l'agent possède des capacités d'interaction avec les autres agents du système. L'agent dispose éventuellement des descriptions des autres agents et il est capable de les influencer via des envois de messages. Une telle architecture est ainsi constituée des fonctions correspondant aux facettes A, I et E. Boissier fait remarquer que la possession de la facette E est optionnelle : « c'est ce qui se produit pour certains agents dont les capacités d'action dans l'environnement sont nulles mais qui possèdent des capacités de communication avec les autres agents » [Boi01]. InteRRaP [Mül96] est un exemple d'architectures d'agents interagissants.
- Une **architecture d'agent social** se caractérise par le fait que l'agent possède des capacités d'interaction et de gestion des relations entretenues avec les autres agents du système. Des entités organisationnelles

2. En interprétant des percepts et en émettant d'autres percepts, il est possible d'établir une forme de communication.

ou équipes sont formalisées et prises en compte par l'agent pour entamer des coopérations. Une telle architecture est ainsi constituée des fonctions correspondant aux facettes A, I, O et E, cette dernière facette étant facultative (comme pour les agents interagissants). DIMA [Gue96] est un exemple d'architecture pour agents sociaux.

3.1.1.3 Autres typologies et architectures *ad hoc*

Il existe des typologies d'architecture d'agents qui sont basées sur d'autres critères que ceux qui précédent. Par exemple, les architectures d'agents sont souvent classées selon les types de décompositions modulaires qui apparaissent par rapport aux fonctions de perception et d'action. Les types les plus connus [Boi01, Mül97] sont :

- **l'architecture horizontale mono-couche** : il y a un seul module et il a accès à la perception et à l'action),
- **l'architecture horizontale avec plusieurs couches** : chacun des modules a « parallèlement » accès à la perception et à l'action,
- **l'architecture verticale modulaire** : il y a plusieurs modules dont un premier qui a accès à la perception et un dernier qui a accès à l'action),
- **l'architecture verticale en couches** : il y a plusieurs modules dont un seul qui a accès à la perception et à l'action.

On peut noter que les architectures verticales sont généralement associées à des agents plutôt délibératifs, alors que les architectures horizontales sont généralement associées à des agents plutôt réactifs.

Autre exemple, Briot [BMP06] propose de classifier différentes architectures d'agents selon la perspective (*i.e.* « *point of view* ») qui est mise en avant par le concepteur. En se basant sur le concept d'architecture logicielle [SG96], il décrit quatre autres types de décompositions architecturales : *selon le cycle d'activation*, *selon les points de vue et traitements associés*, *selon les niveaux et modèles*, et *selon les comportements*. Cette classification permet de faire le rapprochement entre le concept d'architecture d'agents et d'autres concepts issus du génie logiciel (cycles d'activation, points de vue, styles architecturaux, patrons de conception, etc.).

Dans le cas de certains travaux, il est très difficile de classer exactement l'architecture d'agents utilisée. Il est possible que les choix architecturaux du concepteur ne soient pas explicites, notamment quand l'accent est mis sur des préoccupations complètement transversales par rapport aux critères de la typologie considérée. Ainsi, il existe des architectures d'agents qui se caractérisent par des propriétés autres que les capacités de raisonnement, les capacités de coordination ou encore le type de décomposition modulaire. C'est le

cas des architectures qui mettent l'accent sur un découpage fonctionnel des agents ou sur des propriétés extra-fonctionnelles telles que la mobilité, l'apprentissage, ou encore l'adaptation. Par exemple, dans l'architecture d'agents de Garcia et al. [GKdL04], il est possible de définir des agents aux capacités très différentes concernant le type de raisonnement, de coordination, de mobilité, etc. Cette architecture permet aussi bien de concevoir des agents réactifs que délibératifs. Ces agents peuvent être simplement autonomes ou interagissants et leurs degrés de mobilité ou d'apprentissage est soumis à l'appréciation du concepteur car chaque aspect d'un agent se trouve implémenté dans un des modules qui composent l'architecture. Donc, il est inapproprié de classer cette architecture selon les capacités de raisonnement ou selon les capacités de coordination. Lorsqu'un modèle d'agents ne trouve place dans aucune des catégories prévues par une typologie, on l'associe à la catégorie des architectures *ad hoc* (pour cette typologie).

3.1.2 Cas des agents à base de composants

Conformément à ce qui est dit à la sous-section 3.1.1.3, de plus en plus de concepteurs d'agents veulent modeler leur propre architecture d'agents selon les besoins d'une application ou d'un domaine applicatif. Une approche prometteuse pour ce faire consiste à concevoir les agents en composant librement différentes briques, en s'appuyant notamment sur le concept de composant logiciel [Szy98].

Des travaux relativement récents se sont intéressés à la construction d'agents par assemblage de composants. La programmation par composants est en effet intéressante pour ce type de développement car les agents présentent souvent des capacités proches d'une application à une autre (communication, perception, planification, ...). Dans ce cas, la construction d'un agent consiste à *assembler des composants pré-existants qui matérialisent chacun une partie bien spécifique de l'architecture et du comportement de l'agent*. En d'autres termes, un agent est une entité logicielle contenant un assemblage de composants qui matérialise son comportement. Le développement d'applications multi-agents bénéficierait grandement de l'utilisation d'une bibliothèque de composants implantant ces capacités partagées. Plusieurs environnements de développement ont ainsi adopté une telle approche [BJT02, OBDK02, RD02, KMW03].

3.1.2.1 Intérêt des composants pour les agents

Le premier avantage des agents à base de composants est de *permettre à un concepteur de décrire puis gérer de manière unifiée la structure et le*

comportement des agents. Un composant représente une ou plusieurs compétences que possède un agent et qui peuvent être utilisées ou déclenchées sous certaines conditions. Une compétence d'un agent s'apparente donc à un ensemble de fonctionnalités accessibles à travers une interface fournie d'un composant de cet agent. Une compétence offerte par un composant peut requérir d'autres compétences chez les composants de l'agent. Ainsi, le comportement d'un agent se matérialise par un assemblage de composants qui permet de relier entre elles, et de manière structurée, les différentes compétences de cet agent.

Le deuxième avantage des agents à base de composants est de *favoriser la réutilisation et donc d'accélérer le développement et de limiter les erreurs d'implémentation*. Dans les travaux sur la conception d'agents, l'approche à base de composants est avant tout utilisée comme un cadre pour la spécification du comportement de l'agent. En effet, les composants permettent de réutiliser des capacités récurrentes chez les agents (communication, perception, planification, ...). Construire des agents à base de composants consiste principalement à récupérer un ensemble approprié de composants et de les assembler entre eux. Donc, l'utilisation des composants tend à faciliter la construction d'agents. Le fait de réutiliser des composants maintes fois éprouvés limite les risques d'erreurs dûs à une mauvaise implantation.

Le troisième avantage des agents à base de composants est de *faciliter l'adaptation des agents*. Il est assez facile d'ajouter, de supprimer ou de modifier les compétences d'un agent, puisque les applications à base de composants ont l'avantage de pouvoir être ré-assemblées (sans trop d'efforts) selon les principes d'assemblage que définit le modèle de composant utilisé. Par conséquent, les composants logiciels sont utiles non seulement pour la construction d'agents mais aussi pour l'adaptation de ces agents. Notons que le fait de ré-assembler un agent à base de composant permet d'adapter à la fois la structure et le comportement de l'agent.

3.1.2.2 Difficultés des agents à base de composants

Parmi les premiers modèles d'agents qui ont su tirer parti du concept de composants logiciels à des fins d'adaptabilité, on trouve DESIRE [BJT02], MASK [OBDK02], VOLCANO [RD02] et enfin AgentComponent [KMW03, Mei03].

On peut déplorer que ces premiers travaux utilisent des modèles de composants *ad hoc* (voire de simples objets) dont les caractéristiques ne sont pas toujours explicites. En général, la notion de composants dans ces modèles s'apparente (1) soit à des blocs abstraits qui correspondent aux principales fonctionnalités des agents mais dont les interactions ne sont pas explicites,

(2) soit à des blocs concrets (composants *ad hoc* ou objets standards) qui implémentent des fonctionnalités courantes chez les agents logiciels et qui sont reliés entre eux selon une architecture explicite (*i.e.* la structure des agents). MASK et VOLCANO font partie de la première catégorie[3], tandis que DESIRE et AgentComponent font partie de la deuxième catégorie.

Le gros inconvénient de la première catégorie est que le trop haut niveau d'abstraction implique une difficile tâche de développement non seulement lors de la conception mais aussi et surtout lorsqu'on veut réaliser une adaptation. Par exemple, « l'architecture VOLCANO sépare bien les briques, mais remplacer une brique par une autre oblige en général à reprogrammer les adaptateurs liés à cette brique, ce qui déplace en partie la question » [BMP06].

La deuxième catégorie est basée sur des mécanismes de composition concrets. Néanmoins, ces modèles d'agents ne sont que faiblement adaptables car les modifications qui touchent à la structure de l'agent induisent une charge importante de développement, due à l'absence d'un vrai mécanisme d'assemblage. De surcroît, ces adaptations sont complètement intrusives vis-à-vis de la propriété d'autonomie des agents car elles sont déclenchées et réalisées manuellement, donc selon un processus externe aux agents adaptés.

3.1.3 Problématique d'adaptabilité des agents

Nous souhaitons nous concentrer sur l'adaptabilité des agents et des propriétés sur lesquelles l'adaptation a un impact considérable notamment l'autonomie et l'ouverture. Dans la plupart des modèles d'agents, l'adaptation des agents n'est pas spécifiquement traitée. Il est vrai que les agents agissent généralement en fonction du contexte qu'ils perçoivent (percepts issus de l'environnement ou messages issus d'autres agents). Cependant, on peut se demander ce qui se passe lorsque leur contexte évolue de manière imprévisible. Par exemple, saura-t-il avoir un comportement cohérent ou « acceptable » si ses voisins disparaissent inopinément ou si la quantité de messages qu'il reçoit augmente de manière drastique ?

A ce stade de l'étude des architectures d'agents, il est difficile d'évaluer l'impact d'une architecture sur l'adaptabilité des agents. Il faut étudier les travaux les plus significatifs parmi les modèles d'agents auto-adaptables, c'est-à-dire des agents qui prennent en charge toutes les étapes des adaptations qui les concernent, depuis le déclenchement d'une adaptation jusqu'à sa réalisation. Pour comparer ces différents modèles, nous nous appuyons sur

3. Dans MASK et VOLCANO, les agents sont découpés en quatre briques correspondant aux dimensions de l'approche VOYELLES [Dem01].

les principaux critères généraux de l'adaptation définis en section 2.2 (cf. Quoi, Qui, Quand et Comment). Les mécanismes d'adaptation étant moins sophistiqués dans les modèles d'agents que dans les infrastructures d'adaptation détaillées en section 2.4, l'utilisation des cinq critères sur la qualité des adaptations définis en section 2.3.8 (cohérence, performance, disponibilité, simultanéité et ouverture) n'était pas pertinente. Par contre, nous avons utilisé un critère important : la reconfigurabilité de l'infrastructure d'adaptation utilisée par les agents.

Les critères d'adaptabilité des agents sont les suivants :

Portée (quoi ?) : Est-ce que l'adaptation porte sur une partie du comportement (+), sur le comportement dans son ensemble (++) ou sur le comportement et l'architecture de l'agent (+++) ?

Initiateur (qui ?) : Est-ce que l'adaptation est déclenchée par l'agent (+++) ou non (+) ? Si oui, quelle partie de l'agent est responsable de ce déclenchement ?

Moment (quand ?) : Quand est-ce qu'une adaptation peut être déclenchée ? L'adaptation est-elle dynamique (+++) ou non (+) ?

Mise en œuvre (comment ?) : Par quel mécanisme les adaptations sont-elles réalisées ? Est-ce que les mécanismes utilisés sont complètement *ad hoc* (+) ou plutôt élaborés (++) ? La réalisation des adaptations est-elle contrôlée (+++), *i.e.* conforme à la volonté du concepteur de l'agent ?

Reconfigurabilité (meta-adaptation ?) : Est-ce que le mécanisme d'adaptation du modèle d'agents peut s'adapter lui-même (+++) ? Est-il facilement facilement reconfigurable (++) ou pas (+) ?

3.1.4 Synthèse

Grâce aux composants, le concepteur peut plus facilement faire correspondre l'architecture des agents avec leur implémentation. Il peut parfois réutiliser certains composants pour différents agents (par exemple, un composant de communication). Néanmoins, le degré d'adaptabilité de ces modèles d'agents est plutôt faible car la structure des agents reste peu flexible et la plupart des adaptations requièrent l'assistance d'un humain.

Dans les sections suivantes, nous étudions différents modèles [4] d'agents auto-adaptables afin de *déterminer l'impact d'une « architecture d'agents »*

4. Dans la suite du chapitre, nous utilisons le terme « modèle d'agents » pour pouvoir désigner indifféremment les travaux basés sur des architectures classiques (très connues), des architectures spécifiques (moins connues) et aussi les travaux qui n'adoptent pas une architecture fixe.

sur l'« adaptabilité des agents ». Pour ce faire, nous allons distinguer les modèles d'agents qui ont une architecture fixe et ceux qui se basent sur les composants logiciels pour avoir une architecture plus flexible.

3.2 Agents auto-adaptables sans composants

Plusieurs travaux ont eu pour objectif de concilier la problématique d'adaptabilité avec la propriété d'autonomie des agents, en proposant des modèles d'agents auto-adaptables. Dans cette section, nous nous focalisons sur les modèles qui ne se basent pas sur les composants logiciels afin d'estimer par la suite l'impact de l'utilisation de composants logiciels sur l'auto-adaptation.

3.2.1 TouringMachine

Présentation La TouringMachine décrite par Ferguson [Fer92a, Fer92b] est une architecture d'agents hybrides composées de différentes couches. Les agents développés avec la TouringMachine sont destinés à réaliser des tâches de navigation contrainte dans des environnements dynamiques.

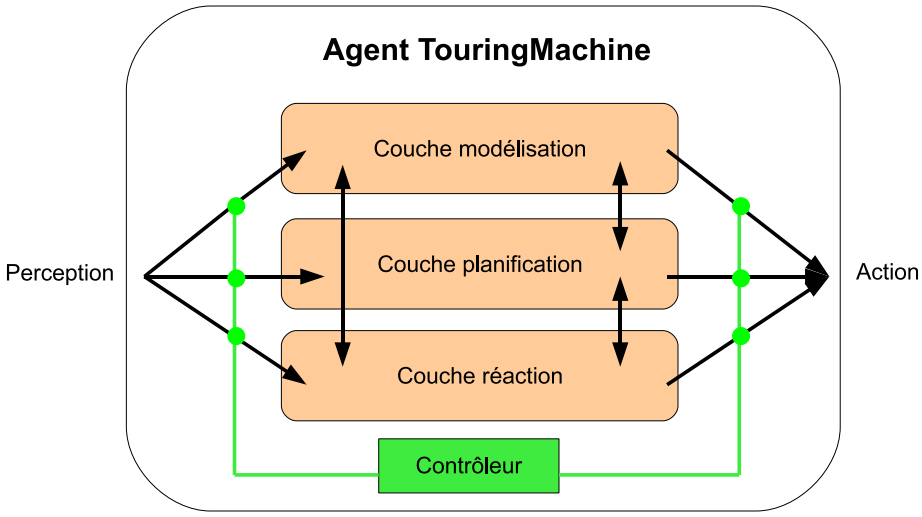

Figure 3.1 – Architecture hybride TouringMachine (vue simplifiée).

Conception d'un agent L'architecture TouringMachine est dîte en couche car elle comporte plusieurs modules : une couche réactive, une couche planification et une couche modélisation (voir la figure 3.1). Cette architecture à trois couches est horizontale car chacune d'elle a accès aux capacités

de perception et d'action de l'agent. La couche réactive permet de réagir à certains événements ou menaces, la couche planification permet de planifier les déplacements de l'agent et la couche modélisation permet de résoudre les éventuels conflits entre les différents buts de l'agent. Les différentes couches interagissent par envoie de messages. Ces interactions sont régulées par un contrôleur de manière à ce que les trois couches induisent un comportement global cohérent.

Adaptation d'un agent La couche modélisation offre la possibilité de modifier les plans de l'agent selon les changements de l'environnement qui ne peuvent pas être traités grâce aux mécanismes de replanification de la couche planification. Par conséquent, la couche modélisation permet à un agent de déclencher des adaptations. Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 3.1.

- *Portée :* une adaptation peut avoir pour cible la couche planification (hors replanification) mais pas la couche réactive, donc elle porte sur une partie du comportement (et aucunement sur l'architecture).
- *Initiateur :* l'agent (par sa couche modélisation) peut déclencher les adaptations.
- *Moment :* les conditions de déclenchement sont spécifiées dans la couche modélisation et portent sur des changements de contexte (*i.e.* les conflits entre les connaissances observées et les connaissances prévues).
- *Mise en œuvre :* la réalisation des adaptations est basée sur un mécanisme d'inférence à base de règles *ad hoc* - spécifiées par le concepteur de l'agent - sur les connaissances prévues et celles qui sont acquises par les couches inférieures.
- *Reconfigurabilité :* l'infrastructure d'adaptation n'est pas prévue pour être facilement reconfigurable.

Portée	**Initiateur**	**Moment**	**Mise en œuvre**	**Reconfigurabilité**
+	+++	+++	+	+

TABLE 3.1 – Critères de l'adaptation pour TOURINGMACHINE

Évaluation L'architecture TOURINGMACHINE fournit un faible degré de contrôle sur les adaptations de l'agent. Dans le cas des systèmes ouverts, les connaissances acquises par l'agent peuvent rapidement diverger des connaissances prévues. Il est donc difficile d'écrire un ensemble de règles d'adaptation puisque les conséquences de chaque règle peuvent être impossibles à prévoir ou trop coûteuses à calculer.

3.2.2 InterRRap

Présentation Comme la TouringMachine de Ferguson [Fer92a], InteRRaP [5] [Mül96] est une architecture d'agents hybrides composées de différentes couches. Les agents développés avec InteRRaP ont été notamment appliqués à la construction de robots de service (pour le transport de cargaison par exemple).

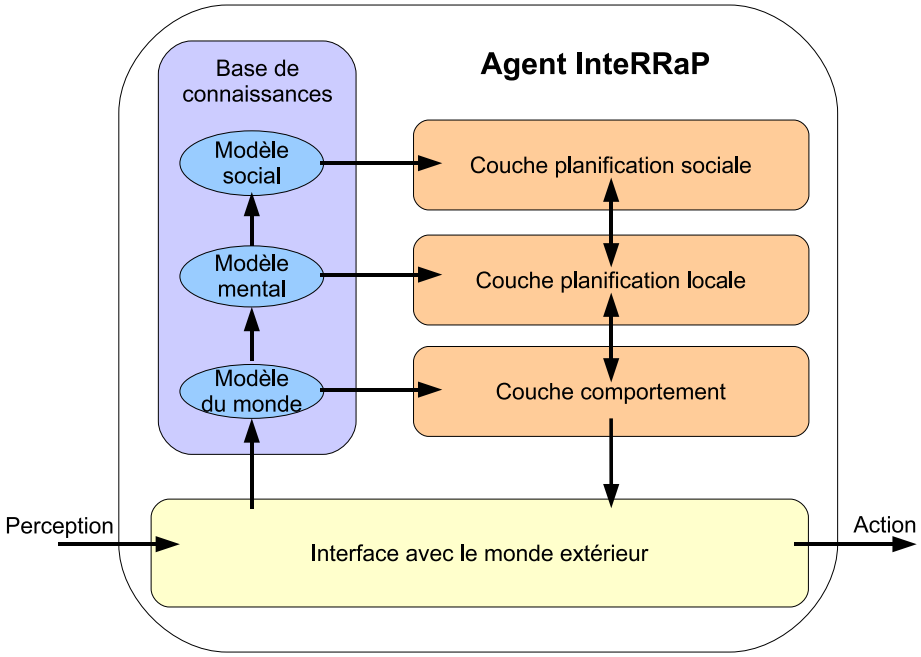

Figure 3.2 – Architecture hybride InteRRaP (vue simplifiée).

Conception d'un agent La conception d'un agent avec InteRRaP requiert la conception des trois modules correspondant à trois couches (comportement, planification, coopération) dans l'architecture. Cette architecture à trois couches est verticale puisque l'accès aux capacités de perception et d'action de l'agent passe par un seul module qui joue le rôle d'interface avec le monde extérieur.

La couche comportement permet de réagir à certains événements ou menaces, la couche planification permet de planifier les déplacements de l'agent et la couche coopération permet de résoudre les éventuels conflits entre les différents buts de l'agent. Chaque couche intègre un cycle de type BDI : activation d'un but, prise de décision et exécution d'une intention. De plus,

5. « *Integration of Reactivity and Rational Planning* ».

chaque couche supérieure prend en compte le fonctionnement des couches inférieures (voir la figure 3.2).

Adaptation d'un agent InteRRaP fournit des mécanismes d'adaptation similaires à TouringMachine, soit la possibilité de modifier les plans de l'agent en cas de nécessité. Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 3.2.

- *Portée :* une adaptation peut avoir pour cible la couche planification (hors replanification) mais pas la couche comportement, donc elle porte indirectement sur une partie du comportement (et aucunement sur l'architecture).
- *Initiateur :* l'agent (par sa couche coopération) peut déclencher les adaptations.
- *Moment :* les conditions de déclenchement sont spécifiées dans la couche coopération et portent sur des changements de contexte.
- *Mise en œuvre :* la réalisation des adaptations est principalement basée sur les mécanismes de contrôle entre couches : si un but ne peut être traité par une couche inférieure alors il est transmis à une couche supérieure et lorsqu'une couche supérieure traite un but alors elle peut affecter en retour une couche inférieure.
- *Reconfigurabilité :* l'infrastructure d'adaptation n'est pas prévue pour être facilement reconfigurable.

Portée	**Initiateur**	**Moment**	**Mise en œuvre**	**Reconfigurabilité**
+	+++	+++	+	+

Table 3.2 – Critères de l'adaptation pour InterRRap

Évaluation L'architecture InteRRaP fournit un faible niveau de contrôle sur les adaptations de l'agent. La qualité des décisions d'un agent soumis à cette architecture est très dépendante de l'environnement et des ressources disponibles. Ainsi, dans les conditions optimales, l'environnement est plutôt calme et l'agent dispose de suffisamment de ressources pour fonctionner principalement selon un mode délibératif et social, puisque les couches supérieures ont la priorité sur les couches inférieures. Par contre, dans un environnement agité et si l'agent dispose de peu de ressources, la couche inférieure de comportement fonctionnera constamment sans faire intervenir les couches supérieures. Dans ce cas, l'agent aura un comportement principalement réactif et aura tendance à négliger son aspect social.

3.2.3 META-CONTROL

Présentation Les agents META-CONTROL, qui ont été introduits par Russell & Wefald [RW91], ont une forme particulière d'architecture en couches horizontales. Cette architecture partage de nombreux points communs avec les architectures hybrides en couches comme la TOURINGMACHINE de Ferguson [Fer92a] ou INTERRAP de Müller [Mül96]. Néanmoins, elle essaie de mieux prendre en compte les ressources disponibles pour établir un bon compromis entre les comportements de haut niveau et les comportements de bas niveau. META-CONTROL est utilisée pour définir des agents rationnels, notamment dans le cadre de jeux (les échecs par exemple).

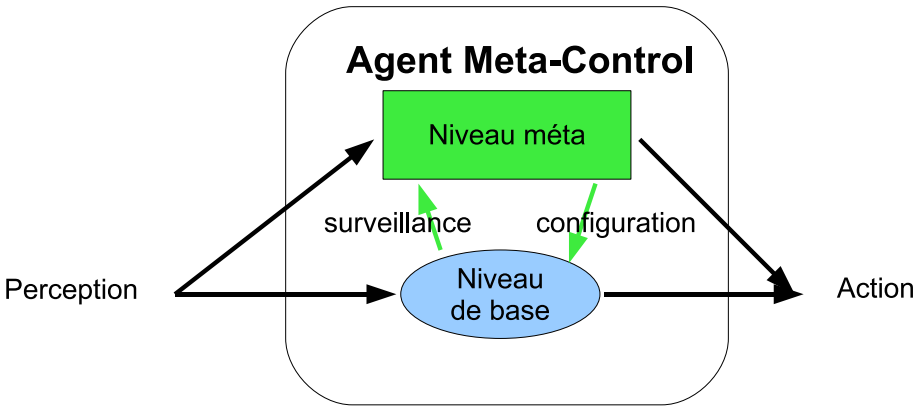

FIGURE 3.3 – Architecture META-CONTROL (vue simplifiée).

Conception d'un agent La conception d'un agent rationnel avec META-CONTROL consiste à résoudre deux problèmes. Premièrement, l'agent doit optimiser son comportement externe selon les ressources disponibles dans l'environnement. Deuxièmement, l'agent doit optimiser ses calculs internes par rapport aux ressources de calculs. Ces deux problèmes sont instanciés à deux niveaux distincts : un *sous-système de niveau objet* qui représente le comportement de planification des actions sur l'environnement et un *sous-système de niveau méta* qui est chargé de surveiller et de configurer le précédent (voir la figure 3.3). Cette architecture à deux couches est horizontale car chacune d'elle a accès aux capacités de perception et d'action de l'agent.

Adaptation d'un agent Dans META-CONTROL, l'adaptation est très différente des architectures hybrides puisqu'elle est fondée sur les liens « méta » entre les deux couches. C'est le niveau méta qui affecte des ressources de calcul aux différentes parties du niveau de base (*i.e.* le niveau objet). Ainsi, en

privant de ressources de calcul certaines parties du niveau de base, le niveau méta peut inhiber des actions du niveau de base. Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 3.3.

- *Portée :* une adaptation peut avoir pour cible le niveau de base, donc elle porte sur le comportement (et pas sur l'architecture).
- *Initiateur :* l'agent (à travers son niveau méta) peut déclencher les adaptations.
- *Moment :* le niveau méta surveille le niveau de base et déclenche des adaptations notamment pour optimiser les ressources matérielles disponibles.
- *Mise en œuvre :* le niveau méta configure le niveau de base, notamment à travers l'attribution des ressources de calcul aux différentes parties du niveau de base.
- *Reconfigurabilité :* le niveau méta est reconfigurable mais uniquement de manière statique, c'est-à-dire en phase de (re)conception.

Portée	**Initiateur**	**Moment**	**Mise en œuvre**	**Reconfigurabilité**
++	+++	+++	++	++

TABLE 3.3 – Critères de l'adaptation pour META-CONTROL

Évaluation Dans l'architecture META-CONTROL, le découplage entre les deux niveaux améliore la séparation des préoccupations (comportement applicatif et comportement d'adaptation), d'où un meilleur degré de contrôle des adaptations. Cependant, la réalisation des interactions entre les deux niveaux repose sur des hypothèses de simplification : (1) les décisions du niveau méta ne sont « que » localement optimales (par souci d'économie de ressources de calcul), (2) les parties du niveau de base concernées par l'affectation de ressources de calcul sont à peu près indépendantes et (3) l'affectation de ressources de calcul se fait au fur et à mesure des besoins (pour tenir compte de certaines dépendances dans le niveau de base). Ces restrictions sont très difficiles à appliquer dans le cas d'agents interagissants (par exemple, des joueurs de football d'une équipe RoboCup [6]) qui doivent se coordonner pour réaliser des actions complexes en temps-réel. L'aspect monolithique du niveau de base rend donc difficile la conception du comportement d'adaptation de l'agent dans le niveau méta, sauf si le comportement applicatif (*i.e.* le niveau de base) est plutôt simple.

6. `http://www.robocup.org/`

3.2.4 Synthèse

Nous avons étudié TouringMachine, InterRRap et Meta-Control, des modèles d'agents auto-adaptables qui ne s'appuient pas sur la notion de composants logiciels. Nous pouvons conclure que l'auto-adaptation d'un agent est possible avec des architectures très classiques telle qu'une architecture délibérative permettant à l'agent de modifier dynamiquement son comportement (par replanification). Mais, ce sont les architectures en couches qui ont mis en œuvre des agents auto-adaptables avec le plus de succès ; bien que les adaptations soient axées sur le comportement et non la structure des agents. En effet, chaque couche représente une préoccupation particulière de l'agent et le mécanisme d'adaptation s'appuie justement sur les relations de contrôle entre les couches. L'avantage de ces architectures à couches est qu'il est assez facile de comprendre la manière dont chaque couche est affectée par des changements contextuels (diminution de ressources, augmentation des sollicitations d'autres agents, etc.).

Concernant les architectures hybrides TouringMachine, InterRRap, on constate que dans les contextes peu favorables, c'est la couche implémentant les comportements de base de l'agent qui est privilégiée par rapport aux couches supérieures. Par ailleurs, une architecture hybride induit des lourdeurs à la conception d'agents aux comportements simples parce qu'il faut obligatoirement définir les différentes couches. De plus, ces architectures souffrent d'un manque de méthodologie de conception claire, notamment lorsque le développeur tente d'établir les priorités de fonctionnement entre les couches d'une architecture.

L'architecture Meta-Control limite cet inconvénient en centralisant la préoccupation d'adaptation dans un niveau méta qui surveille et configure le niveau de base. Mais, l'aspect monolithique des deux niveaux de cette architecture rend très difficile la conception d'agents ayant un comportement complexe.

Pour finir, notons qu'aucune de ces architectures ne permet à l'agent de modifier sa structure interne afin par exemple d'ajouter ou de supprimer des fonctionnalités. De plus, la réutilisation d'une couche d'architecture pour concevoir un agent différent implique généralement de nombreux changements dans le reste de l'architecture, ce qui rend ce genre d'opération totalement impraticable.

3.3 Agents auto-adaptables à base de composants

Dans cette section, nous analysons les principaux travaux qui proposent des modèles d'agents capables de s'auto-adapter et qui sont basés sur la notion de composants logiciels.

3.3.1 MAST

Présentation Le modèle MAST [Ver04], développé par l'équipe SMA de l'Ecole des Mines de Saint-Etienne, est un environnement de développement et de déploiement d'applications multi-agents. Dans MAST, une application multi-agent est construite comme un assemblage de composants. La conception, qui reprend la décomposition VOYELLES [Dem01], s'applique sur deux niveaux et le modèle de composant proposé tient compte de ces niveaux en distinguant deux types de composants. Au *niveau système*, un SMA est composé d'agents (vus comme des composants) et de certaines entités fonctionnelles nommées composants orientés système (COS) : par exemple, la plateforme FIPA [FIP] comme intergiciel de communication et le kernel MadKit [GF01] comme support organisationnel. Au *niveau agent*, un agent est également conçu comme un assemblage de composants au sens plus classique du terme, dits composants orientés agent (COA).

Conception d'un agent Pour construire un agent MAST, il suffit de lui ajouter un composant « noyau » et un ensemble de composants fonctionnels (les COAs), comme le montre la figure 3.4. MAST permet de connecter automatiquement et de manière flexible les composants qui forment un agent. En effet, les événements émis par les différents composants de l'agent sont capturés par le *composant noyau*. Ce dernier se base sur la description sémantique de chaque composant (*i.e.* leur rôle) pour transmettre chaque événement reçu aux composants capables de le traiter. L'évènement est transmis aux COAs - successivement selon l'attribut « priorité » qui les caractérise - jusqu'à ce que l'évènement soit consommé par l'un des COAs. Pour définir un assemblage de composants, le concepteur doit définir la priorité (un réel) de chaque interface des composants présents.

Adaptation d'un agent Ces connexions implicites sont intéressantes pour la flexibilité qu'elles procurent à l'assemblage de l'agent, puisqu'elles sont automatiquement déduites en fonction des interfaces des composants. Ainsi,

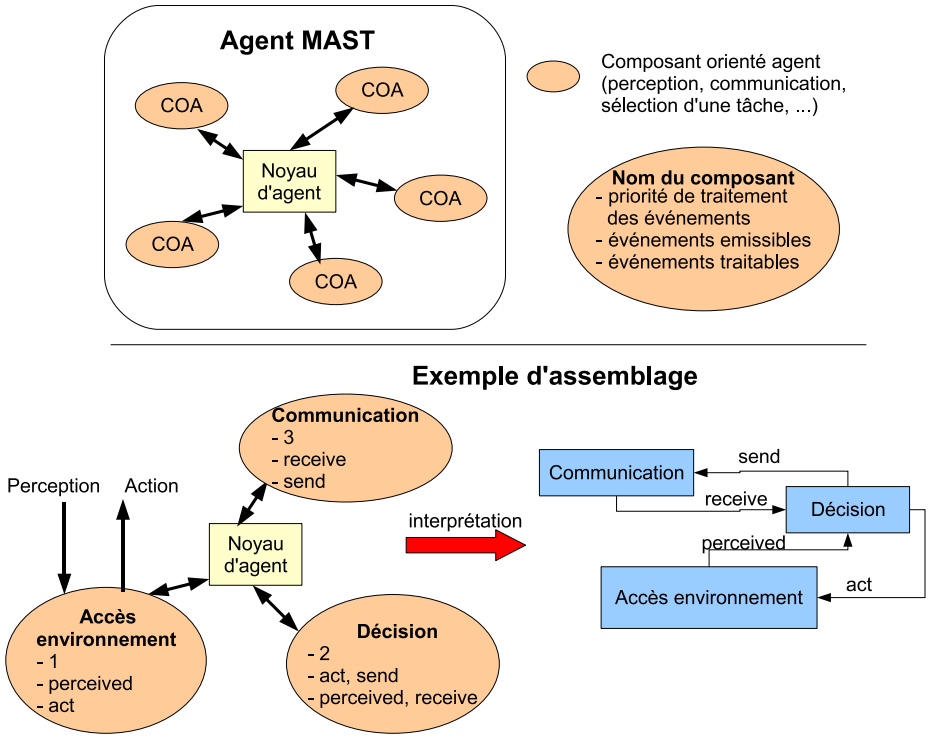

FIGURE 3.4 – Exemple d'architecture d'agent autonome avec MAST.

l'adaptation des agents qui consiste à lui ajouter ou supprimer des composants est réalisée automatiquement. Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 3.4.

- *Portée :* la cible de l'adaptation est un assemblage de composants, donc elle porte à la fois sur le comportement et l'architecture de l'agent.
- *Initiateur :* c'est le développeur de l'agent qui déclenche les adaptations, lorsqu'il ajoute ou supprime un composant de cet agent ; donc, l'agent n'est pas complètement auto-adaptable.
- *Moment :* le développeur de l'agent déclenche dynamiquement les adaptations par l'ajout ou la suppression de composants.
- *Mise en œuvre :* la réalisation des adaptations s'appuie sur un mécanisme de bus logiciel chargé de la distribution d'événements en se fondant sur le « *matching* » entre interfaces de communication des composants.
- *Reconfigurabilité :* l'infrastructure d'adaptation n'est pas reconfigurable car pour modifier l'assemblage induit par un ensemble de composants, il faut modifier les priorités spécifiés dans les interfaces des composants.

Portée	**Initiateur**	**Moment**	**Mise en œuvre**	**Reconfigurabilité**
+++	+	+++	++	+

TABLE 3.4 – Critères de l'adaptation pour MAST

Évaluation Le mécanisme de connexion automatique et flexible de MAST souffre de trois inconvénients majeurs : (1) chaque agent possède un goulot d'étranglement en son composant « noyau » à cause du modèle de transmission en étoile, (2) il est impossible de déclencher une adaptation autrement qu'en ajoutant ou en supprimant manuellement des composants, et (3) le manque de contrôle sur l'assemblage de composants accroît le risque d'introduire des erreurs dans le comportement d'un agent. En effet, il n'est pas évident de prévoir l'impact de chaque modification de l'assemblage sur le comportement global de l'agent, en particulier lorsque des composants complètement nouveaux sont ajoutés. Ainsi, la flexibilité de l'assemblage dans MAST peut se faire au détriment de l'explicitation du comportement de l'agent, car le concepteur de l'agent n'a pas l'occasion de spécifier avec précision les conditions d'assemblage qu'il souhaite.

3.3.2 MAGIQUE

Présentation MAGIQUE [RMS01, Sec03] est un framework multi-agent permettant de construire des SMAs où les agents sont organisés hiérarchiquement. Les SMAs produits s'apparentent à un système d'objets concurrents répartis dans lequel l'organisation hiérarchique des agents permet à chaque agent d'invoquer des fonctionnalités quelle que soit leur localisation grâce au principe de délégation des tâches. En effet, pour chaque invocation, l'agent essaie de traiter la tâche concernée s'il possède la compétence nécessaire. Dans le cas contraire, il essaie de faire traiter la tâche par l'un de ses voisins directs ou de ses subalternes, ou alors il délègue la tâche à son supérieur hiérarchique. Ce mécanisme de délégation permet aux agents de faire appel à des compétences sans préciser de destinataires.

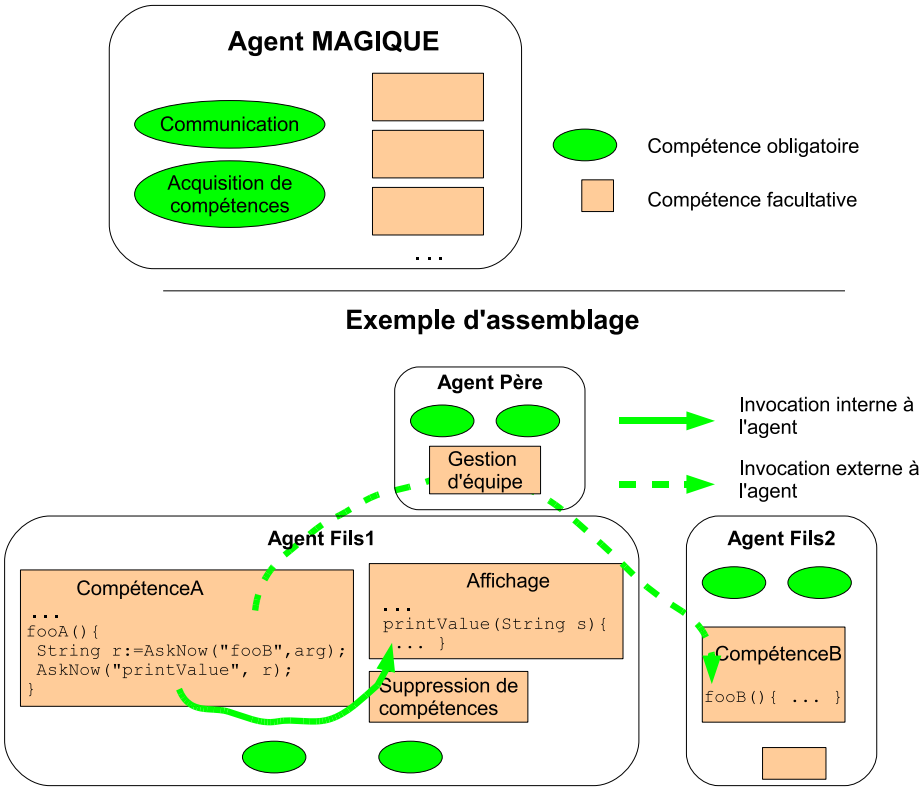

FIGURE 3.5 – Exemple d'architecture d'agent social avec MAGIQUE.

Conception d'un agent Un agent est vu comme une coquille contenant des *compétences*. Chaque compétence est un « ensemble cohérent de fonctionnalités » permettant de réaliser une tâche précise qui est regroupée dans

un composant logiciel [7]. Donc, les compétences peuvent être développées indépendamment des agents qui les utiliseront. Pour invoquer une opération offerte par une compétence interne ou externe à l'agent, il faut utiliser la primitive [8] `perform(''nomOperation'', arguments)` sans besoin de nommer le destinataire, ce qui peut poser problème quand des opérations sémantiquement différentes ont la même signature dans une hiérarchie d'agents. Pour créer un agent, il faut ajouter un ensemble de compétences à sa « coquille vide », en particulier une compétence pour communiquer et une compétence pour pouvoir acquérir d'autres compétences (voir la figure 3.5 qui montre un exemple social [9]). Enfin, si l'on veut que l'agent soit proactif, alors il doit posséder une *compétence d'action* (qui implémente la méthode `action()`) qui représente le cœur du comportement de l'agent.

Adaptation d'un agent Un agent MAGIQUE peut acquérir (ou oublier) et invoquer dynamiquement de nouvelles compétences. Par exemple, si un agent A_1 utilise beaucoup une compétence d'un agent B_2 alors il peut demander à B_2 de lui envoyer cette compétence plutôt que de continuer à l'utiliser « à distance ». Cet échange est réalisé sans difficulté grâces aux compétences de communication et d'acquisition de compétences dont disposent les agents. Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 3.5.

- *Portée :* la cible de l'adaptation est un assemblage de composants, donc elle porte à la fois sur le comportement et l'architecture de l'agent. Il s'agit d'acquérir ou d'oublier une compétence, ou encore de remplacer la compétence d'action.
- *Initiateur :* le déclenchement de l'adaptation peut être manuel (par le développeur de l'agent) ou automatique, mais principalement pour optimiser le système multi-agent.
- *Moment :* le déclenchement d'une adaptation de manière statique ou dynamique (*i.e.* l'acquisition ou la suppression d'une compétence) dépend du comportement de l'agent et de la tâche courante.
- *Mise en œuvre :* la réalisation des adaptations est basée sur l'utilisation conjointe des compétences de communication et d'acquisition de

7. L'implémentation n'est pas fondée sur un modèle de composants au sens strict de la définition de Szyperski [Szy02] mais plutôt sur des objets faciles à déployer et à composer.

8. `perform` invoque une compétence sans retourner de résultat. Lorsqu'un résultat doit être retourné, on utilise plutôt `AskNow` (invocation synchrone), `ask` (invocation asynchrone) ou `concurrentAsk` (invocation concurrente).

9. Le caractère social de ces agents se justifie par l'aspect hiérarchique de l'organisation du SMA dans MAGIQUE : chaque agent est supervisé par un chef et il peut superviser plusieurs agents.

compétences, mais il y a peu de précision sur la spécification des déclenchements d'adaptations.
– *Reconfigurabilité :* l'infrastructure d'adaptation utilisée (*i.e.* les compétences fondamentales) n'est pas facilement reconfigurable dans la mesure où les spécifications des adaptations sont mélangés dans le code applicatif.

Portée	Initiateur	Moment	Mise en œuvre	Reconfigurabilité
+++	+++	+++	++	+

TABLE 3.5 – Critères de l'adaptation pour MAGIQUE

Évaluation Dans le modèle MAGIQUE, un agent est principalement vu comme un fournisseur de compétences pour les autres agents du SMA. Ainsi, l'adaptation d'un agent par échange de compétence tient plus du fait d'une optimisation du déploiement des compétences au sein du SMA que des besoins propres à cet agent. C'est pourquoi, ce modèle n'offre pas de mécanisme pour spécifier du comportement d'adaptation de l'agent de manière découplée par rapport au comportement applicatif de l'agent : tout doit être codé dans la compétence d'action. En outre, si une compétence n'est disponible chez aucun agent, alors ce sont tous les agents qui utilisent cette compétence qui sont paralysés. Pour éviter ce genre de problème, le concepteur du système doit essayer d'assurer que chaque compétence soit répliquée chez plusieurs agents.

3.3.3 MALEVA

Présentation MALEVA [BMP06, MB01, GHM+99, Lhu98] est un modèle d'agents proposé par le LIP6 à l'Université de Paris 6, notamment pour la conception de systèmes multi-agents [10]. Ce modèle est utilisé pour construire des agents logiciels dont le comportement et la structure peuvent être adaptés. Il prévoit deux sortes d'interaction entre composants : un *flot de données* et un *flot de contrôle*. Le principal avantage d'avoir un flot de contrôle explicite dans un assemblage de composant est de faciliter l'identification de problème de concurrence. MALEVA a notamment été utilisé dans les domaines des sociétés artificielles et de la simulation multi-agent [BM06].

Conception d'un agent Avec MALEVA, un concepteur d'agents dispose d'une bibliothèque de composants représentant des comportements élémen-

10. Le modèle de composant associé à ce modèle d'agents s'appelle également MALEVA.

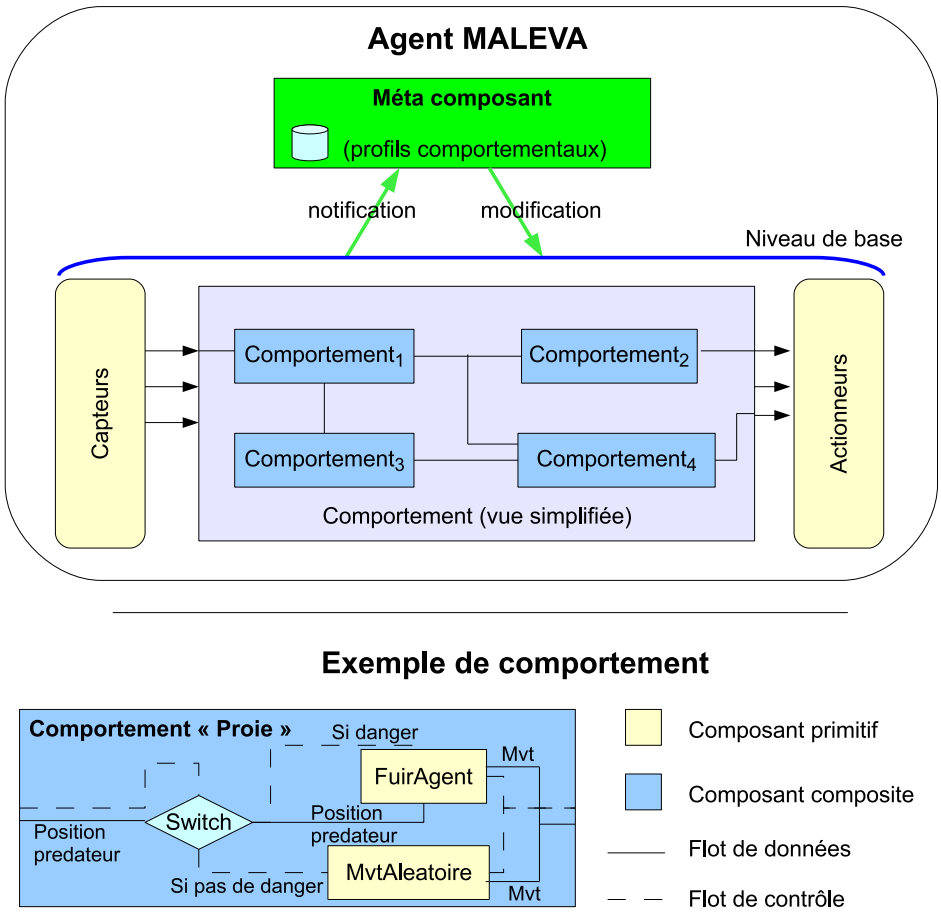

FIGURE 3.6 – Exemple d'architecture d'agent réactif avec MALEVA.

taires. Il peut assembler plusieurs composants existants dans un composant composite de manière à constituer des comportements complexes réutilisables dans différents contextes ou différentes familles d'agents. Les auteurs utilisent aussi une forme simplifiée de patron de conception [GHJV95] permettant d'instancier un comportement complexe abstrait et spécialisable : un pré-câblage entre des composants fixes et des composants à insérer. Concevoir un agent revient à composer, de manière structurelle et fonctionnelle, différents composants comportementaux au sein de l'agent, comme l'illustre la figure 3.6. Par exemple, le comportement « Proie » est défini en connectant trois composants : la fuite (le composant `FuirAgent`), le mouvement aléatoire (le composant `MvtAléatoire`), et un composant de contrôle appelé `Switch` qui permet de réifier la conditionnelle perception/non-perception. La gestion du contrôle revient à définir et ajouter des composants spécifiques qui ont à charge de traiter les compositions associées aux données et celles associées aux contrôles.

Adaptation d'un agent Les adaptations peuvent être déclenchées par l'agent après qu'un de ses composants de comportement émette une requête particulière. Cette requête est transmise à un composant spécifique appelé « méta composant », qui se comporte comme un fournisseur de « profils comportementaux » préalablement définis par le concepteur de l'agent. Ces profils prédéfinis décrivent des assemblages spécifiques de composants. Donc, ils améliorent le contrôle du concepteur de l'agent sur les assemblages à générer en interdisant les assemblages non prédéfinis. Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 3.6.

- *Portée :* la cible de l'adaptation est un assemblage de composants, donc elle porte à la fois sur le comportement et l'architecture de l'agent.
- *Initiateur :* l'agent (par ses composants comportementaux) peut déclencher les adaptations.
- *Moment :* le code de chaque composant comportement prévoit d'émettre une requête particulière (déclenchement d'adaptation dynamique) en fonction de l'état (et des messages entrants) du composant.
- *Mise en œuvre :* un méta-composant reçoit les demandes d'adaptation (flot méta) et adapte l'agent selon un des profils comportementaux préalablement définis par le concepteur de l'agent.
- *Reconfigurabilité :* l'infrastructure d'adaptation (*i.e.* le méta-composant) n'est pas facilement reconfigurable car le choix des adaptations est spécifié dans le code du méta-composant.

Portée	Initiateur	Moment	Mise en œuvre	Reconfigurabilité
+++	+++	+++	++	+

TABLE 3.6 – Critères de l'adaptation pour MALEVA

Évaluation Le principal inconvénient de MALEVA vient du fait que l'adaptation d'agents est basée sur un mécanisme *ad hoc*. L'interaction entre le méta-composant et les composants comportementaux est définie de manière très « opérationnelle ». Ainsi, le concept de profil comportemental n'est pas réifié et ses relations avec le méta composant et les composants comportementaux ne sont pas explicites. L'évolution d'un agent (ajout/retrait de composants ou de profils comportementaux) requiert de réimplanter systématiquement la partie de l'agent qui est en charge de l'adaptation, *i.e.* le méta composant. Ainsi, la charge de développer un méta-composant qui implémente toutes les évolutions possibles d'un agent revient entièrement au concepteur de cet agent. La façon dont sont définis les profils comportementaux des agents n'est pas explicitée, ce qui conduit au fait que les opérations de ré-assemblage de composants doivent être implantées profil par profil. Ainsi, au lieu de définir plusieurs profils comportementaux et de proposer un mécanisme capable d'adapter un agent selon n'importe quel profil, il est nécessaire d'implanter plusieurs opérations de ré-assemblage dans un méta-composant, chacune de ces opérations représentant un profil comportemental. Le fait que ce soit un composant comportemental qui fasse appel au composant méta en est une parfaite illustration.

3.3.4 BOND

Présentation BOND [BM00, BKB+04, BM05] est un framework d'agents et d'objets distribués qui a été développé par l'Université de Purdue aux États-Unis. Il a été le premier système d'agents basés sur Java à offrir un haut degré d'adaptabilité. Les auteurs définissent la notion d'agent *mutable*, c'est-à-dire un agent qui est auto-adaptable ou qui peut être adapté par une entité externe. Le modèle BOND a été appliqué pour concevoir des agents fortement adaptables et mobiles, notamment dans le cadre de systèmes hautement hétérogènes tels que les réseaux *ad hoc*, les workflows adaptatifs ou les services multimédia.

Conception d'un agent La couche agent de BOND est une extension du framework agent JADE[11]. L'architecture d'un agent est spécifiée par Blueprint, qui un langage déclaratif conçu pour décrire la structure interne des

11. JADE webpage. `http://sharon.cselt.it/projects/jade/`.

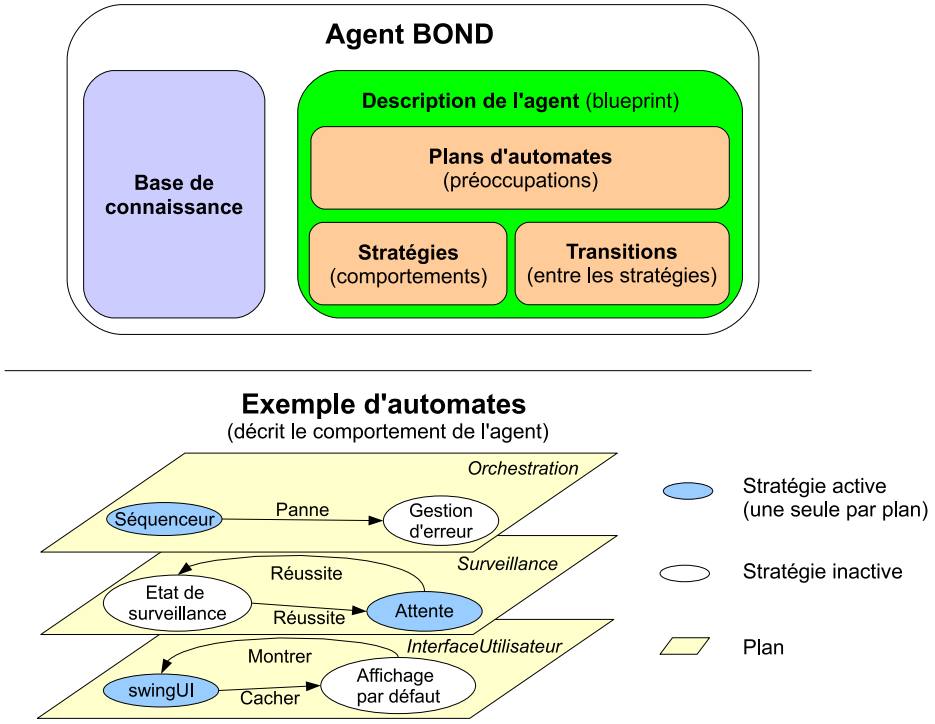

FIGURE 3.7 – Exemple d'architecture d'agent adaptable avec BOND.

agents BOND. Les principaux composants [12] d'un agent BOND sont une base de connaissances, un ensemble de plans d'automates, un ensemble d'états ainsi que les transitions entre ces états (voir la figure 3.7). Les transitions dans BOND représentent un changement de « stratégie », chaque stratégie étant un comportement caractérisé par des informations méta (pré-conditions post-conditions, équivalence avec un autre stratégie, ressources matérielles requises, ...). Les plans de l'automate multi-plans expriment des activités parallèles, c'est-à-dire une exécution simultanée des diagrammes état-transition.

Adaptation d'un agent Pour adapter dynamiquement des agents, les auteurs introduisent une technique de mutation appelée « chirurgie [13] d'agents » (« *agent surgery* »). Cette technique décrit les mutations comme des séries d'opérations primitives dans un automate multi-plan. Les différentes opéra-

12. L'implémentation n'est probablement pas un modèle de composants au sens strict de la définition de Szyperski [Szy02] mais plutôt une structure à objets qui est modifiable grâce à un langage de description. Ce mécanisme de modification s'apparente très bien à un réassemblage de composants.

13. Ce terme, qui fait référence à une intervention sur un organisme vivant, correspond à ce que nous appelons adaptation dynamique dans le reste de cette thèse (voir le chapitre 2).

tions primitives sont : *l'ajout ou la suppression d'un état ou d'une transition, le remplacement de la stratégie d'un état* (*e.g.* passer de l'utilisation d'une interface graphique à l'utilisation de lignes de commandes lors d'une migration de l'agent sur une machine dépourvu d'environnement graphique), *le découpage d'une transition en ajoutant un état intermédiaire* (*e.g.* synchronisation interagent ou débogage), *la suppression d'un état intermédiaire dans une transition* (*e.g.* suppression du mode débogage), *l'ajout et la suppression de plans* (*e.g.* ajouter, remplacer ou supprimer une fonctionnalité telle que le contrôle à distance, le mécanisme de raisonnement ou la négociation), *la fusion ou la fission d'agents* (*e.g.* fusionner des agents de contrôle pour avoir un contrôle unifié et optimiser la communication et la mémoire utilisée dans le système, fissionner un agent pour appliquer des priorités différentes à ses fonctionnalités), et *la réduction d'agents* (*e.g.* supprimer avant une migration les états et les transitions par lesquels on ne doit pas ou plus passer). Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 3.7.

- *Portée :* s'agissant de composants, la cible de l'adaptation est à la fois le comportement et l'architecture de l'agent.
- *Initiateur :* l'agent (ou une entité logicielle extérieure) peut déclencher les adaptations.
- *Moment :* le déclenchement des adaptations est exprimé par les transitions de l'automate multi-plans, donc selon les besoins applicatifs pendant l'exécution.
- *Mise en œuvre :* l'utilisation du langage de description d'agents (Blueprint) permet de spécifier les opérations d'adaptation. La réalisation des adaptations dynamiques n'est pas explicitée mais vu que BOND est implémenté en Java, ces adaptations peuvent utiliser l'introspection et notamment les classes du package `java.lang.reflect`.
- *Reconfigurabilité :* l'automate multi-plan peut être adapté avec le langage Blueprint ; une évolution du modèle pour plus de réflexivité (*e.g.* plan d'un automate qui est adapté selon un autre plan) fait partie des améliorations envisagées par les auteurs de BOND.

Portée	**Initiateur**	**Moment**	**Mise en œuvre**	**Reconfigurabilité**
+++	+++	+++	++	++

TABLE 3.7 – Critères de l'adaptation pour BOND

Évaluation Les auteurs distinguent d'une part, la mutabilité faible qui est assez courante dans les systèmes d'agents et d'autre part, la mutabilité forte qui est plus rare et difficile. La mutabilité faible est le fait d'adapter un agent

à des changements mineurs. Autrement dit, on peut modifier (typiquement, il s'agit d'étendre) quelques fonctionnalités ou appliquer un patch de sécurité, mais le rôle de l'agent dans le système ne change pas. Par contre, la mutabilité forte consiste à modifier la structure et le comportement d'un agent de manière radicale, c'est-à-dire que ses spécifications peuvent changer complètement. BOND est un des rares modèles à traiter ces deux sortes d'adaptations. Toutefois, l'absence de distinction entre le comportement applicatif et le comportement adaptatif nuit à la compréhension de l'agent, à son évolution et donc à la réutilisabilité du code d'un agent dans d'autres applications.

3.3.5 JavAct$^{\delta}$

Présentation JavAct$^{\delta}$ est un modèle d'agents proposé par l'équipe IRIT-LYRE de l'Université Paul Sabatier de Toulouse pour la programmation d'applications concurrentes, réparties et mobiles [LA06, Ler06, LA07]. Ce modèle [14] dérive du précédent JavAct 4 [15] [AMM01] et de l'architecture réflexive proposée par Marcoux et al. [MMMS01]. Les auteurs s'attachent à permettre au concepteur de vérifier l'architecture d'un agent à partir d'une description d'assemblage de composants, afin de générer des agents flexibles mais toujours corrects. Le modèle de composants utilisé est proche d'EJB [16] : un composant est caractérisé par son interface, une ou plusieurs réalisations et un moyen d'exécution (son conteneur).

Conception d'un agent Chaque agent possède un niveau de base contenant du code fonctionnel (comportement déterminé par le programmeur de l'application agent) et un niveau méta décrivant les services non fonctionnels comme l'envoi asynchrone de messages, la création locale et distante d'agents, le changement de comportement, la mobilité, la gestion de la boîte aux lettres et la gestion du cycle de vie de l'agent (voir la figure 3.8). Ces derniers sont appelés les « mécanismes opératoires de l'agent » et chacun d'eux est représenté par un composant à grain fin (ou *micro-composant*) qui offre un unique service non fonctionnel. En outre, les agents ont une architecture de micro-composants en étoile autour d'un connecteur nommé Contrôleur. Ce dernier fonctionne comme un bus logiciel et permet d'implémenter le principe de délégation pour faciliter l'évolution de l'architecture. Par ailleurs, le niveau

14. Le sigle delta représente la capacité d'adaptation dynamique du modèle.
15. Une bibliothèque Java standard qui s'appuie sur les concepts d'acteur et d'implémentation ouverte.
16. http ://java.sun.com/products/ejb

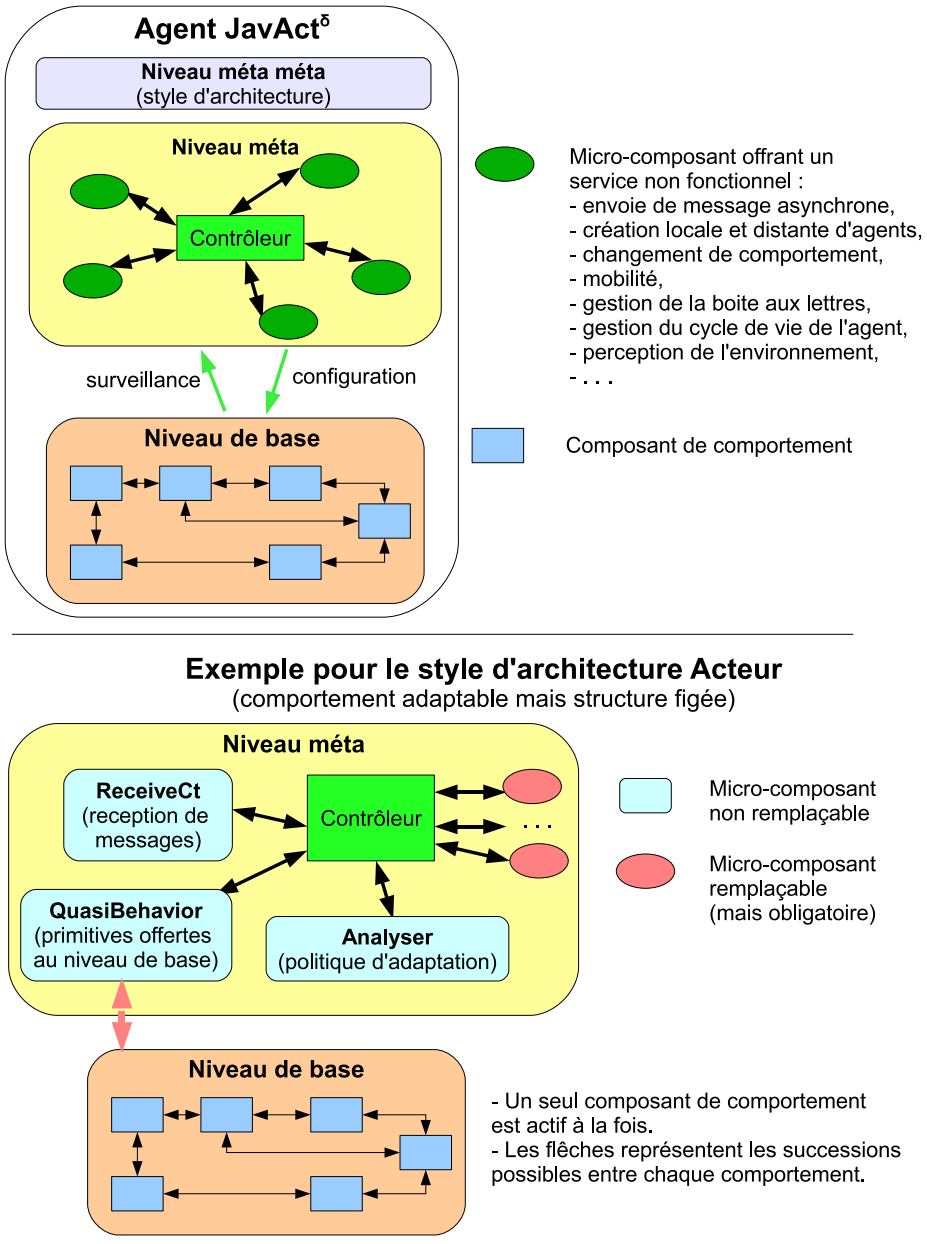

FIGURE 3.8 – Exemple d'architecture d'agent adaptable avec JAVACT$^{\delta}$.

méta de l'agent correspond à un type d'agent défini selon un « style d'architecture » (niveau méta méta). JavAct$^{\delta}$ permet de concevoir différents styles d'agents (réactif, BDI, mobile, ...), celui qui est explicité un style axé sur le *modèle d'acteurs* [Hew77], sur l'adaptabilité et la mobilité.

Adaptation d'un agent Dans ce modèle, on distingue les adaptations au niveau de base et les adaptations au niveau méta. Une adaptation au niveau de base consiste à remplacer le composant de comportement courant par un autre grâce à une primitive du type `void become(QuasiBehavior b)`. Une adaptation au niveau méta consiste à remplacer un micro-composant grâce à une primitive `with(Type newComp)` pour remplacer le micro-composant de type `Type` par `newComp`. Ces primitives sont également accessibles par le niveau de base. Les adaptations sont dynamiques et pour éviter les problèmes d'incohérence, l'adaptation n'est immédiate que si l'agent est en attente de message, sinon elle est différée après la fin de l'exécution du comportement courant. Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 3.8.

- *Portée :* la cible de l'adaptation est un assemblage de composants, donc elle porte à la fois sur le comportement et l'architecture de l'agent.
- *Initiateur :* l'agent peut déclencher les adaptations depuis son niveau de base (de manière autonome) ou depuis son niveau méta (à la demande du concepteur).
- *Moment :* le déclenchement automatique des adaptations est fonction d'une part, du contexte d'exécution qui est obtenu d'après un composant de perception et d'autre part, de la politique d'adaptation implémentée dans le composant spécifique (`Analyser`).
- *Mise en œuvre :* le niveau méta de l'agent est dynamiquement adaptable grâce à l'utilisation de la réflexion [Smi82], tandis que le niveau de base de l'agent est dynamiquement adaptable grâce au modèle d'acteur.
- *Reconfigurabilité :* le niveau méta de l'agent (constituée d'un ensemble de micro-composants) est reconfigurable mais uniquement de manière statique, c'est-à-dire en phase de (re)conception.

Portée	**Initiateur**	**Moment**	**Mise en œuvre**	**Reconfigurabilité**
+++	+++	+++	+++	++

Table 3.8 – Critères de l'adaptation pour JavAct$^{\delta}$

Évaluation Ce modèle vise avant tout à permettre l'adaptation du fonctionnement des agents, c'est-à-dire de leurs activités non-fonctionnelles, en

particulier pour leur permettre de s'adapter à leur environnement d'exécution. Cette approche se caractérise par l'utilisation exclusive pour le méta-niveau de micro-composants qui offrent un service non fonctionnel unique. Cela permet d'avoir un niveau méta qui soit configurable, contrairement à JavAct 4. De plus, un formalisme de vérification assure que les modifications de l'architecture - au niveau méta - sont correctes, grâce à un ADL (langage de description d'architecture) *ad hoc*. Néanmoins, un agent n'est pas complètement réflexif dans le modèle d'agent adaptable dans la mesure où certains micro-composants du niveau méta sont non remplaçables : le composant pour la réception de messages, le composant qui implémente la politique d'adaptation et le composant qui implémente les primitives offertes au niveau de base (il sert d'interface entre le niveau de base et le niveau méta).

3.3.6 Synthèse

Dans cette sous-section, nous avons étudié plusieurs modèles d'agents auto-adaptables à base de composants : MAST, Magique, MALEVA, BOND et JavAct$^{\delta}$.

Dans chacun des travaux ci-dessus, le modèle de composants utilisé est spécifique au modèle d'agents [17]. D'autre part, les auteurs mettent clairement en avant la forte adaptabilité des agents grâce à leur modèle d'agents. Les adaptations d'un agent sont facilitées par l'utilisation de composants. Mais, la problématique d'assemblage est transversale aux composants de l'agent car elle peut concerner l'ensemble des composants. C'est pourquoi, chacun des modèles prévoit un processus d'adaptation basé sur des composants et des mécanismes spécifiques. Chaque modèle met en avant certaines caractéristiques :

- la facilité à ajouter ou supprimer des composants : MAST, Magique, JavAct$^{\delta}$;
- l'implication de plusieurs agents lors d'une adaptation : Magique, BOND ;
- la facilité à attribuer une succession de comportements prédéfinis (a fortiori, des assemblages de composants) à un agent : MALEVA ;
- la capacité d'appliquer uniformément des changements mineurs ou majeurs à un agent : BOND ;
- la possibilité de faire évoluer le modèle d'agents en garantissant sa cohérence : JavAct$^{\delta}$.

17. Notons que pour Magique et BOND, il ne s'agit pas de composants qui respectent la définition que nous avons adopté [Szy02], notamment car il n'ont pas d'interfaces requises explicites.

3.4 Bilan sur les modèles d'agents auto-adaptables

Il existe différents modèles d'agents. Certains modèles sont purement théoriques, d'autres ont une implémentation de référence et d'autres ont même un environnement de développement dédié. La multiplicité des modèles d'agents, l'absence d'implémentation de certains modèles et les liens trop forts qui relient des modèles d'agents à un environnement de développement particulier rendent très difficile une comparaison objective et complète des modèles d'agents. Par exemple, il n'existe pas de taxonomie permettant de choisir un modèle d'agents en fonction du domaine d'application visé. En revanche, chaque modèle d'agents possède des propriétés particulières visant à favoriser un sous-ensemble des caractéristiques classiques des agents telles que : l'autonomie, la sociabilité, la réactivité, la pro-activité, la coopération, la négociation, la mobilité, etc.

Deux pistes radicalement différentes ont été engagées pour s'écarter des approches classiques et de leurs inconvénients :

1. Certains concepteurs ont cherché à mélanger différents types d'architectures classiques pour concilier différentes caractéristiques et ainsi regrouper le maximum d'avantages au sein d'une même architecture (par exemple, les architectures hybrides).
2. D'autres concepteurs ont cherché à définir des architectures hautement configurables en s'appuyant sur la notion de composants logiciels.

3.4.1 Principales caractéristiques des travaux étudiés

Nous avons étudiés différents modèles d'agents auto-adaptables. Certains étaient basés sur la notion de composants logiciels et d'autres non (voir le tableau 3.9).

	Portée	**Initiateur**	**Moment**	**Mise en œuvre**	**Reconfigurabilité**
Modèles d'agents sans composants					
TouringMachine	+	+++	+++	+	+
InteRRap	+	+++	+++	+	+
Meta-Control	++	+++	+++	++	++
Modèles d'agents à base de composants					
MAST	+++	+	+++	++	+
Magique	+++	+++	+++	++	+
MALEVA	+++	+++	+++	++	+
BOND	+++	+++	+++	++	++
JavAct$^{\delta}$	+++	+++	+++	+++	++

Table 3.9 – Comparatif des modèles d'agents auto-adaptables

Pour avoir des agents qui intègrent correctement différentes propriétés, il faut définir des modèles d'agents composites plutôt que des modèles monoli-

thiques. C'est pourquoi, de nombreuses architectures d'agents ont été développées, en particulier les architectures en couches. Parmi ces dernières, des architectures hybrides (comme TouringMachine et InterRRap) ou méta (comme Meta-Control) permettent d'avoir des agents auto-adaptables. Ils proposent des mécanismes pour permettre aux agents d'adapter efficacement leurs comportements en cas de nécessité. Néanmoins, aucune de ces architectures ne permet à l'agent de modifier sa structure interne afin par exemple d'ajouter ou de supprimer des fonctionnalités.

Aujourd'hui, de plus en plus de concepteurs d'agents veulent définir des architectures plus spécifiques en s'appuyant notamment sur le concept de composant logiciel. L'étude de l'état de l'art montre que l'utilisation d'un modèle d'agents à base de composants permet d'avoir des agents auto-adaptables qui peuvent être modifiés dynamiquement. De plus, une adaptation peut concerner l'ensemble des composants, donc elle peut modifier en profondeur à la fois le comportement et la structure de l'agent, ce qui est particulièrement appréciable dans les SMAs ouverts.

Certains des mécanismes permettant d'adapter automatiquement un assemblage de composants posent problème, en particulier vis-à-vis de la propriété d'autonomie des agents. Dans le cas de MAST, les composants de l'agent sont automatiquement réassemblés à condition que le concepteur ajoute ou retire des composants. Ces opérations d'adaptation sont souvent simples mais parfois le concepteur peut être amené à reconfigurer chaque composant présent pour pouvoir insérer un nouveau composant à l'endroit souhaité. De même, Magique se focalise sur la simplification de l'ajout et du retrait d'un composant sauf que ce sont les agents qui ont l'initiative de ces adaptations, notamment dans une perspective de répartition de charge au niveau d'un système multi-agent (échange de compétences).

L'idéal à atteindre est un modèle d'agents capables de déclencher et de réaliser automatiquement des adaptations en fonction de leurs besoins, tout en maintenant un contrôle sur les modifications à effectuer. Il est intéressant de pouvoir spécifier en quelque sorte quelles sont les adaptations qui sont autorisées (à la manière des profils comportementaux de MALEVA, de l'ensemble fini d'opérations primitives de mutation des agents de BOND ou du mécanisme de vérification a posteriori des adaptations de JavAct$^{\delta}$) de manière à être sûr que les adaptations automatiques permettrons un comportement cohérent des agents.

De plus, les capacités d'adaptation des agents doivent être définies au niveau modèle à travers des concepts de haut niveau (*e.g.* configuration et politique) qui favorisent notamment la séparation des préoccupations, la réutilisabilité et la maintenance des agents. Dans les modèles d'agents actuels, l'utilisation de composants permet au moins de mettre en évidence

les éléments clés de l'infrastructure d'adaptation utilisée (à la manière des composants de contrôle de MALEVA et de JavAct$^{\delta}$). Ainsi, les stratégies de déclenchement et de contrôle de ces adaptations sont plus explicites et configurables par le concepteur de l'agent.

Enfin, la possibilité pour l'infrastructure d'adaptation d'un modèle d'agents de s'adapter elle-même - ou en tout cas d'être facilement personnalisable - constitue un atout capital : le modèle d'agents auto-adaptables pourra convenir à un grand nombre de domaines d'applications ou de contextes d'exécution. JavAct$^{\delta}$ est une illustration de cette capacité de configurabilité du modèle d'agents.

3.4.2 Discussion

Aucune architecture d'agents n'est optimale. Nous pensons que *dans un environnement ouvert, les agents doivent pouvoir s'adapter souvent et de la manière la plus autonome qui soit*. Notre étude dans ce chapitre tend à montrer que les agents à base de composants sont plus flexibles que les agents standards et que l'emploi de composants logiciels permet d'adapter simplement le comportement et la structure des agents. Les meilleurs modèles d'agents auto-adaptables sont ceux qui offrent une infrastructure d'adaptation simple à utiliser, riche en expressivité et surtout elle doit faire partie intégrante du modèle d'agents afin que les adaptations soient non intrusives vis-à-vis de la propriété d'autonomie des agents. Ainsi, au lieu de créer une architecture d'agents figée et complexe, il est préférable que l'architecture de l'agent soit facilement adaptable et réutilisable.

Cependant, dans la plupart des modèles d'agents auto-adaptables les mécanismes de spécification et de réalisation des adaptations manquent d'*abstraction*. Peu de modèles proposent un langage déclaratif pour spécifier le comportement d'adaptation de l'agent. De plus, ils supposent que le concepteur des agents connaît exactement le comportement des composants qu'il veut utiliser. Et, quand ce n'est pas le cas, la spécification des adaptations est simple mais pas assez précise ou riche pour définir des adaptations complexes.

Par ailleurs, même avec des composants logiciels, la conception d'éléments réutilisables dans un agent reste un problème compliqué. Pour les agents à base de composants, la problématique de *réutilisabilité* est simplement reporté sur le concepteur de composants. En effet, comme un composant peut être utilisé dans des agents qui sont structurés de différentes manières, le concepteur de composants doit doter ses composants d'interfaces les plus génériques possibles. De plus, il n'est pas certain qu'on puisse réutiliser ces composants lorsqu'on change le domaine applicatif des agents, notamment par

ce que les contraintes extra-fonctionnelles (par exemple, les ressources matérielles) peuvent changer. Il est donc souhaitable d'utiliser des composants et des mécanismes d'adaptation qui intègrent ces aspects extra-fonctionnels.

Un autre problème mal étudié est celui de la *réutilisation de comportements complexes d'un agent*, par exemple un assemblage de plusieurs composants. La majorité des travaux se focalisent uniquement sur la façon d'ajouter ou de supprimer un simple composant. Mais, peu de travaux se sont intéressés à permettre d'utiliser dans un agent une partie de comportement qui est issu d'un autre agent, par exemple pour étendre ou spécialiser un comportement déjà éprouvé dans un contexte différent. Certains travaux font état de l'utilisation de composants composites, ce qui limite leur approche aux modèles de composants hiérarchiques.

Enfin, dans tous les travaux que nous avons vu, le modèle d'agents proposé s'adresse soit à un concepteur qui veut utiliser une des architectures d'agents classiques (réactif, hybride, etc.), soit à un concepteur qui veut utiliser une architecture originale et flexible (à base de composants). Dans le premier cas, l'architecture est figée car les préoccupations du concepteur d'agents sont fixes (par exemple, la réactivité, la robustesse, l'optimalité du comportement ou le mode d'interaction entre agents) tandis que dans le second cas, l'architecture est évolutive. Cependant, les évolutions architecturales sont très largement guidées par des besoins comportementaux (ajout ou suppression des fonctionnalités offertes par un composant). Selon nous, il faut aller plus loin dans le *découplage entre le comportement applicatif de l'agent et l'architecture de l'agent*. Dans un système ouvert, il serait intéressant de permettre à l'agent de modifier son architecture aussi bien pour des raisons comportementales que pour des raisons extra-fonctionnelles telles que la variabilité des ressources matérielles disponibles (CPU, mémoire, énergie, bande passante, etc.).

Deuxième partie

Contribution

Chapitre 4

MadCar : un modèle de moteurs d'assemblage

L'adaptation est le processus par lequel un logiciel est modifié afin de prendre en compte un changement [KBC02a], que ce soit au niveau de l'environnement ou du logiciel lui-même. Il s'agit d'un processus en trois temps. Il faut *détecter* les changements, *décider* de la réaction la plus appropriée à la situation détectée, et enfin *réaliser* les traitements décidés (voir le chapitre 2). L'adaptation est dite *dynamique* si ce processus est réalisé sans arrêter l'exécution du logiciel. Cette dynamicité s'avère nécessaire dans différents domaines d'application (médecine, finances, télécommunications, ...) où un arrêt peut être très coûteux financièrement ou dangereux du point de vue humain ou environnemental. Elle est également requise pour concevoir des application autonomes capables de s'auto-adapter.

Nous nous intéressons dans ce chapitre aux étapes de décision et de réalisation des adaptations dynamiques dans les applications à base de composants [Szy02]. Dans ce contexte, la réalisation d'une adaptation se traduit par une *reconfiguration*[1] de l'assemblage de composants constituant l'application. Cette reconfiguration consiste à ajouter, supprimer ou remplacer des composants ou des connexions entre composants. La reconfiguration de l'application englobe également la modification des attributs des composants et a fortiori le transfert d'état entre composants lors de ces reconfigurations.

Dans ce chapitre, nous introduisons MadCar[2] un modèle de moteurs dédiés à l'adaptation dynamique et automatique d'applications à base de composants. Ce modèle fournit une solution uniforme, basée sur un solveur de contraintes, permettant non seulement la construction d'applications par

1. Parfois, nous employons le terme de « (ré)assemblage » pour désigner à la fois la construction d'un assemblage et la reconfiguration de cet assemblage.

2. « Model for Automatic and Dynamic Component Assembly Reconfiguration ».

assemblage automatique mais aussi la reconfiguration dynamique de ces applications. La spécification des adaptations est à la fois globale à l'application et découplée des composants. De ce fait, l'approche MADCAR présente l'avantage d'éviter les problèmes de cohérence des adaptations rencontrés dans les approches où chaque composant s'adapte indépendamment des autres [DL03, PBJ98]. De plus, les spécifications d'assemblage et d'adaptation peuvent être réutilisées avec des jeux de composants différents, et ce pour des composants fournissant des contrats sur les ressources matérielles (CPU, mémoire, énergie) qu'ils requièrent. Enfin, comme notre approche d'adaptation est indépendante d'un modèle de composants particulier, nous proposons un formalisme qui permet au concepteur de l'application de définir l'état de l'application et de spécifier les règles de transfert d'état de manière générique. Hormis l'existence de certaines interfaces extra-fonctionnelles (contrats sur les ressources matérielles, gestion de l'état et de l'activité d'un composant), aucune hypothèse n'est faite sur les composants ou sur le modèle de composants utilisé.

Le reste de ce chapitre se décompose de la manière suivante. La section 4.1 introduit le concept de moteur d'assemblage et identifie des critères pour évaluer ce genre d'infrastructure d'adaptation. La section 4.2 décrit les différentes parties du modèle MADCAR. La section 4.3 traite plus particulièrement de la gestion de l'état d'une application pendant les réassemblages. La section 4.4 décrit différents travaux liés à l'adaptation d'application à base de composants. Enfin la section 4.5 fournit un résumé de notre approche.

4.1 Besoins pour les moteurs d'assemblage

L'automatisation de la tâche d'assemblage nécessite un *moteur d'assemblage* qui peut fonctionner à la place d'un humain. Ce moteur doit construire automatiquement des applications en assemblant des composants. De plus, le besoin d'adapter dynamiquement ces applications nécessite que l'assemblage doit pouvoir être réalisé sans stopper les applications.

Dans cette section, nous introduisons le concept de moteur d'assemblage. Puis, nous proposons des critères pour mesurer la capacité d'un moteur d'assemblage (ou d'un infrastructure d'adaptation similaire) à tenir compte des besoins en automatisation et en dynamicité.

4.1.1 Définition d'un moteur d'assemblage

Partant des définitions existantes du problème d'assemblage automatique [SMBV03, IT02], nous définissons le concept de moteur d'assemblage

comme suit : *étant donné un ensemble de composants et une description d'application, un* moteur d'assemblage *permet d'assembler un sous-ensemble des composants disponibles pour construire une application qui satisfait la description fournie.* Un modèle abstrait de moteur d'assemblage est illustré par la figure 4.1. Un moteur d'assemblage abstrait nécessite quatre entrées :

- Un ensemble arbitraire de *composants* peut être utilisé pour construire un nouvel assemblage. Les composants assemblés peuvent éventuellement correspondre à différents modèles de composants, par souci de généralité. Cependant, dans le cadre de notre travail, nous supposons que tous les composants appartiennent au même modèle de composants.
- La *description d'application* correspond à une spécification des fonctionnalités de l'application et de ses propriétés extra-fonctionnelles. Comme pour l'entrée des composants, la description d'application qui peut être prise en charge par un moteur d'assemblage est censée être dépendant du domaine d'application visée. Lorsque cette description est vide, le moteur d'assemblage est libre de réaliser un assemblage arbitraire en se basant sur les informations contenues dans les composants et sur la politique d'assemblage.
- Une *politique d'assemblage* est une description (partielle) de comportement du moteur. Cette entrée garantit l'ouverture du moteur d'assemblage, ce qui fait référence à la capacité de personnaliser/contrôler le processus de (ré)assemblage (par exemple, en fonction du contexte d'exécution).
- Le *contexte d'exécution* [CCDG05] fait référence à différentes informations (par exemple, la consommation du CPU ou la quantité de mémoire disponible) dont les valeurs doivent être considérées pour prendre de bonnes décisions d'assemblages et dont les changements sont susceptibles de déclencher un réassemblage de l'application. En cours d'exécution de l'application, les étapes de déclenchement et de décision du processus d'assemblage sont notamment basées sur le contexte d'exécution de l'application. Par exemple, si la quantité de mémoire disponible diminue alors le moteur d'assemblage peut décider de reconstruire l'assemblage en utilisant des composants plus légers.

4.1.2 Critères d'évaluation

Pour évaluer et comparer des moteurs d'assemblages, nous définissons les critères suivants.

1. Degré de cohérence : il fait référence à la capacité du moteur d'assemblage à empêcher l'introduction involontaire d'erreurs de fonctionne-

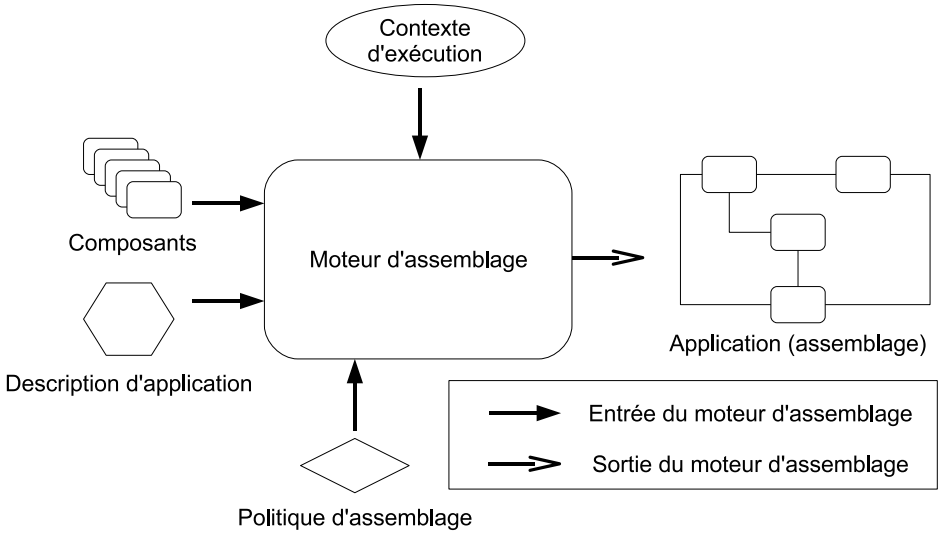

FIGURE 4.1 – Entrées et sortie d'un moteur d'assemblage abstrait.

ment dans l'application à cause de processus de (ré)assemblage.

2. Degré de performance : il fait référence à la capacité du moteur d'assemblage à empêcher la diminution de performance de l'application pendant un (ré)assemblage, c'est-à-dire de garantir que la consommation de ressources du processus d'assemblage est suffisamment faible pour ne pas priver l'application pendant les (ré)assemblages.

3. Degré de disponibilité : il fait référence à la capacité du moteur d'assemblage à garantir l'utilisation de certains services de l'application pendant le processus d'assemblage.

4. Degré de simultanéité : il fait référence à la capacité du moteur d'assemblage à coordonner des adaptations qui sont déclenchées ou réalisées simultanément.

5. Degré d'ouverture : il fait référence à la capacité du moteur d'assemblage à permettre la personnalisation de son comportement.

Les critères 1, 2, 3, 4 représentent différentes propriétés nécessaires au moteur d'assemblage pour réassembler « sans nuisance » une application qui s'exécute. Le critère 5 caractérise la réutilisabilité d'une telle infrastructure d'adaptation.

4.2 Modèle MADCAR

MADCARest un modèle de moteurs permettant l'assemblage et l'adaptation automatiques d'applications à base de composants logiciels. Les adaptations visées vont d'une simple modification de la valeur d'un attribut de composant jusqu'à un remplacement complet d'un assemblage de composants.

4.2.1 Hypothèses de travail

MADCAR établit une séparation entre les spécifications d'applications et leurs implémentations, à la manière de ce que préconise l'approche MDA [3] (« *Model Driven Architecture* ») [BJRV04]. Cela permet de spécifier des applications à base de composants sans connaître le modèle concret qui sera utilisé pour les composants et leurs contrats, à la différence des approches plus spécialisées comme Chang et al. [CC07]. Les spécifications d'assemblage que nous proposons ont vocation à être réutilisées telles quelles pour des jeux de composants issus de différents modèles de composants.

Pour permettre cette séparation, les composants doivent satisfaire quelques hypothèses.

1. Les composants doivent être *homogènes*, *auto-documentés* et ils doivent fournir des interfaces extra-fonctionnelles pour *gérer leur état et leur activité*. Par composants homogènes, nous désignons des composants se conforment au même modèle de composants. L'auto-documentation fait référence aux contrats. Les composants doivent avoir des contrats qui décrivent les attributs paramétrables, les interfaces fournies ou requises, et les ressources matérielles requises pour leur fonctionnement (par exemple, les consommations en CPU, mémoire et énergie). Ainsi, on peut prendre en compte dans MADCAR des contraintes extra-fonctionnelles lors des adaptations. Cette approche est également applicable aux composants qui ne fournissent aucun contrat sur les besoins matériels, notamment lorsque le domaine de l'application ne les utilise pas. Mais, dans ce cas, les décisions d'adaptations ne portent que sur la spécification fonctionnelle de l'application et leur pertinence au contexte.
2. Dans la version actuelle de MADCAR, on s'attache à adapter une application ou un fragment d'application situé sur *un seul site d'exécution*. L'adaptation d'applications distribuées sort du cadre de ce chapitre.

3. C'est pour cette raison que les trois lettres M, D et A sont mises en valeur dans l'acronyme MADCAR.

3. Nous ne nous intéressons dans ce travail qu'à la *composition horizontale de composants* car elle est applicable à n'importe quel modèle de composants. Notre approche permet l'utilisation de composants composites, mais elle ne prend pas en charge la composition hiérarchique des composants composites. Ce point permet d'abstraire des spécificités de chaque modèle de composants (voir la sous-section 4.2.2) pour pouvoir traiter - *i.e.* assembler et réassembler - des composants à travers un processus uniforme.

4.2.2 Moteur d'assemblage

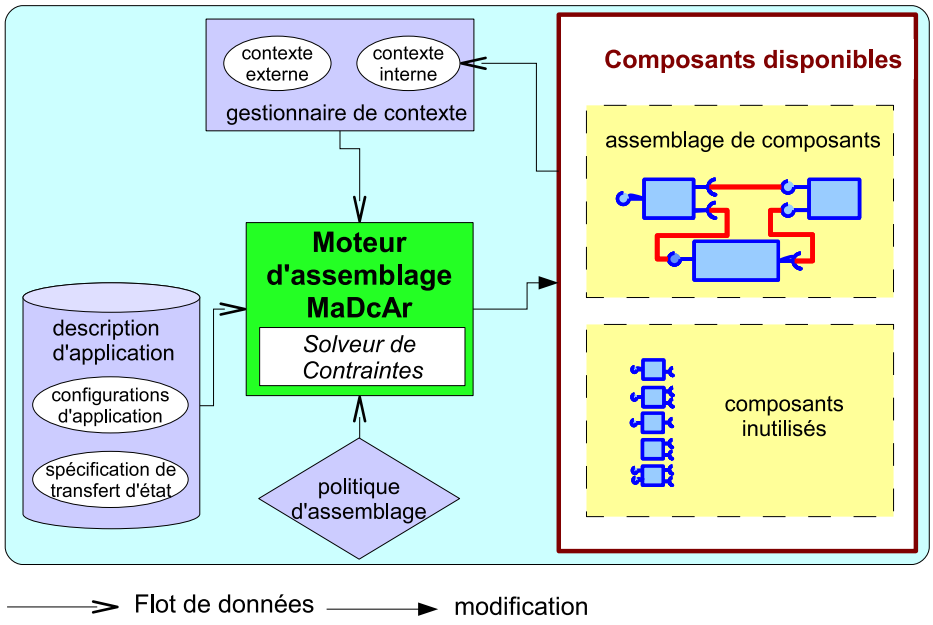

FIGURE 4.2 – Entrées-Sorties de MADCAR.

Comme illustré dans la figure 4.2, un **moteur d'assemblage** MADCAR permet de construire ou d'adapter une application à partir de quatre entrées : un *ensemble de composants* à assembler, une *description de l'application* qui représente la spécification des assemblages valides en termes de propriétés fonctionnelles et extra-fonctionnelles, une *politique d'assemblage* qui dirige les décisions d'adaptation et un *contexte* qui reflète aussi bien les informations sur les ressources matérielles (CPU disponibles, bande passante, etc.) que l'état de l'application (*i.e.* valeurs des attributs des composants assemblés).

Les composants, la description de l'application et le moteur d'assemblage peuvent être réutilisés pour construire des applications différentes. Nous sui-

vons une *approche par « framework »* : un moteur d'assemblage MADCAR a une partie générique et une partie qui est spécifique au modèle de composants considéré. Pour prendre en charge un nouveau modèle de composants, il faut implémenter un petit ensemble d'opérations typiques sur les composants : connexion/déconnexion de deux composants, activation/désactivation d'un composant et importation/exportation de l'état d'un composant.

La description d'application regroupe un ensemble de configurations alternatives et une spécification pour le transfert d'état. La description d'application et la politique d'assemblage sont spécifiées en termes de contraintes. Ainsi, un (ré)assemblage d'application avec MADCAR se traduit par un problème de satisfaction de contraintes (CSP [4], [Kum92, LP01, RLP00]).

4.2.3 Gestionnaire de contexte

Généralement, la définition d'un *contexte d'exécution* consiste à spécifier un ensemble de sondes logicielles qui peuvent fournir des « données pertinentes », c'est-à-dire des valeurs qui doivent être utilisées pendant un processus. En fait, « *Le Contexte n'est pas simplement l'état d'un environnement prédéfini avec un ensemble fixe de ressources d'interaction. Il fait partie d'un processus d'interaction avec un environnement en perpétuel changement composé de ressources reconfigurables, migrables, distribuées et multi-échelles* » [CCDG05]. Par conséquent, la première entrée d'un moteur d'assemblage MADCAR est un **gestionnaire de contexte** qui modélise l'évolution d'un contexte d'exécution. Le gestionnaire de contexte doit définir un ensemble de sondes utilisées par un moteur d'assemblage non seulement pour affecter les décisions d'adaptation mais aussi pour déclencher les adaptations. De plus, il permet de définir la fréquence à laquelle les données contextuelles sont mises à jour et si nécessaire la taille de l'historique des valeurs mesurées par chaque sonde.

Les sondes du gestionnaire de contexte peuvent être en rapport avec les aspects internes ou externes de l'application.

- Le *contexte externe* d'une application inclut les informations sur les ressources matérielles (par exemple, CPU et mémoire), les réseaux disponibles (par exemple, bande passante courante et bande passante maximale) et des données géophysiques (par exemple, localisation et température).
- Le *contexte interne* d'une application est constitué des informations issues des composants de l'application : quels composants sont utilisés, quelles sont les connexions entre les composants et quel est l'état

4. « *Constraint Satisfaction Problem* ».

courant de l'application (valeurs des attributs de l'assemblage de composants).

Pour les exemples utilisés dans ce chapitre, les sondes doivent fournir au minimum les niveaux courants pour la CPU disponible, la mémoire libre et l'énergie (pour une batterie) disponible sur l'infrastructure où sont déployés les composants. Les valeurs de ces ressources matérielles seront respectivement notées $value_{cpu}$, $value_{memory}$ et $value_{energy}$.

À titre d'illustration, la figure 4.3 montre quelques spécifications de sondes.

- La sonde `networkAvailabilitySensor` teste périodiquement si le réseau ethernet est disponible en évaluant une fonction `#isAvailable`. Cette sonde alimente le contexte externe de l'application.
- La sonde `bufferSizeSensor` récupère régulièrement la taille courante d'un attribut `buffer` depuis l'état de l'application (voir la sous-section 4.3.1).

Dans cet exemple, chaque ressource observée possède sa propre période de mise à jour.

```
networkAvailabilitySensor := Sensor new.
networkAvailabilitySensor resource : 'external/hardware/network/ethernet'.
networkAvailabilitySensor operation : #isAvailable updatePeriod : 1000.
...
bufferSizeSensor := Sensor new.
bufferSizeSensor resource : 'internal/stateTransferNet/bufferNode'.
bufferSizeSensor operation : #getSize updatePeriod : 500.
```

FIGURE 4.3 – Exemple de spécification de contexte.

La conception d'un gestionnaire de contexte pose des problèmes typiques de performance, par exemple lorsqu'il y a plusieurs sondes qui doivent se mettre à jour très souvent. De plus, les données brutes collectées par les sondes doivent généralement être propagées et regroupées pour en retirer des informations de plus haut niveau et détecter des situations contextuelles utiles à l'application. Ces questions sortent du cadre de cette thèse. Dans la suite, nous supposons que les valeurs fournies par le gestionnaire de contexte utilisé sont toujours à jour lorsqu'elles sont lues par le moteur d'assemblage. Cette succincte description d'un gestionnaire de contexte est suffisante pour illustrer notre processus de réassemblage automatique. Pour plus de détails concernant la modélisation de contextes, on peut se référer aux travaux qui décrivent des « *frameworks* » génériques de contexte [CRS07, DL05, DSA01].

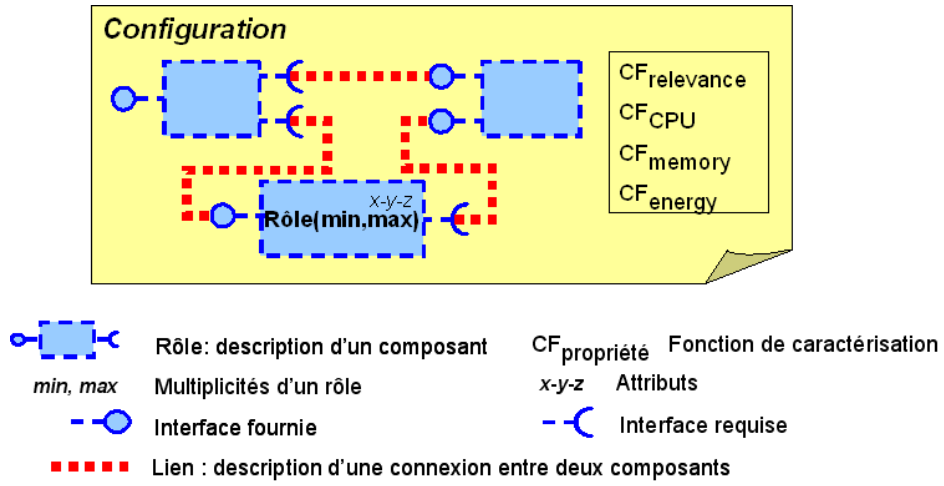

FIGURE 4.4 – Configuration dans MADCAR.

4.2.4 Spécification des assemblages valides

Une description d'application dans MADCAR définit principalement l'ensemble des assemblages valides (i.e. autorisée par le concepteur pour cette application) sous forme d'un ensemble de configurations alternatives. De plus, elle définit de façon transverse la manière de maintenir l'état de l'application lors des réassemblages, par transfert d'état entre un ancien assemblage et un nouvel assemblage. Une **configuration** MADCAR décrit une *famille d'assemblages* similaires car ayant les mêmes contraintes structurelles. En effet, chaque configuration se compose d'un graphe de *rôles* et d'un ensemble de *fonctions de caractérisation*, comme le montre la figure 4.4. Selon les définitions de leurs fonctions de caractérisation, chaque configuration est plus ou moins appropriée aux différentes situations contextuelles qui peuvent survenir pendant l'exécution de l'application [5].

4.2.4.1 Rôles et liaisons

Un **rôle** R est une description de composant contenant un ensemble de *contrats* [Mey92, BJPW99]. Ces contrats spécifient au moins :

– un ensemble d'interfaces (requises ou fournies) qui symbolisent les interactions possibles avec d'autres rôles ;

5. Inversement, si le concepteur d'application ne prévoit pas de configurations suffisamment diverses (par négligence ou par nécessité), alors certaines situations contextuelles ne seront pas prises en compte de manière optimale ou pas prises en compte du tout, ce qui peut se traduire par un fonctionnement de l'appli en mode dégradé.

- un ensemble d'attributs dont les valeurs permettent d'initialiser les composants ;
- deux multiplicités qui définissent les nombres minimum (min) et maximum (max) de composants qui peuvent remplir simultanément ce rôle [6].

Le concepteur peut utiliser des contrats supplémentaires dans un rôle afin de contraindre le choix d'un composant avec la précision souhaitée, notamment en fonction de caractéristiques techniques (par exemple, la taille du composant). Grâce aux rôles, aucune référence directe aux composants de l'application n'est introduite par les configurations.

Pour permettre l'initialisation automatique des composants sélectionnés pour un rôle, chaque attribut est associé à une fonction d'initialisation. De plus, un attribut est déclaré **variable** si sa valeur peut être modifiée par le composant correspondant. Il est déclaré **fixe** dans le cas contraire. Les valeurs des attributs variables font l'objet d'un transfert d'état. Ces différents aspects sont détaillés dans la section 4.3.

Un composant est dit *compatible avec un rôle* lorsqu'il satisfait tous les contrats du rôle. Une configuration est dite *éligible* lorsque chaque rôle de la configuration peut être rempli par un nombre de composants qui est au moins égal à la multiplicité minimale du rôle.

Un assemblage de composants se construit à partir d'une configuration. Pour ce faire, chaque composant est sélectionné pour remplir un des rôles de cette configuration et les composants doivent être connectés entre eux selon les *liaisons* spécifiées entre les rôles qu'ils occupent. Dans MADCAR, les **liaisons entre rôles** ont une sémantique unique et ne se traduisent au niveau des composants que par une référence directe entre une interface fournie et une interface requise appartenant à deux composants distincts. Il est important de remarquer qu'un rôle est unique non seulement par ses interfaces et ses attributs, mais aussi par ses liaisons avec les autres rôles. Il en découle que *les composants qui remplissent un même rôle doivent être connectés exactement aux mêmes composants*. Cette règle permet de facilement automatiser la construction des assemblages à partir d'une configuration.

Par exemple, l'assemblage résultant du lien entre $R_a(1,3)$ et $R_c(1,1)$ consiste à relier tous les composants jouant le rôle R_a au composant jouant le rôle R_c. Concernant les multiplicités des rôles, si le modèle de composants utilisé ne permet pas de connecter une interface fournie (resp. requise) à plusieurs interfaces requises (resp. fournies), alors les configurations dont certains rôles ont une multiplicité minimale strictement supérieure à 1 ne pourront pas être utilisées. Pour les autres configurations, il n'y aura qu'un composant par rôle.

6. Si besoin, nous noterons un rôle $R(min, max)$, où $min \geq 0$ et $max \geq min$.

4.2.4.2 Fonctions de caractérisation

Chaque configuration est caractérisée à travers des propriétés extra-fonctionnelles. Ces propriétés sont utilisées par le moteur d'assemblage afin de choisir une configuration lors d'un réassemblage. Pour chaque propriété P, le concepteur de configuration doit implémenter une **fonction de caractérisation** CF_P qui mesure P. Pour que les propriétés de configuration soient comparables entre elles, un concepteur doit définir des fonctions de caractérisation normalisées. Les résultats doivent être compris entre 0 et 100.

La propriété la plus essentielle est la pertinence de la configuration pour un contexte donné (notée `relevance`). À celle-là, on peut ajouter des propriétés qui traduisent d'autres préoccupations, notamment les coûts d'utilisation pour la configuration. Dans cette thèse, nous prenons en considération : le coût CPU d'une configuration (noté `CPU`), le coût mémoire d'une configuration (noté `memory`) et le coût en énergie d'une configuration (noté `energy`).

- $CF_{relevance}$ mesure la pertinence d'une configuration par rapport à une situation contextuelle, c'est-à-dire à la fois les valeurs des sondes (contexte externe) et l'état de l'application (contexte interne).
- CF_{energy}, CF_{memory}, CF_{cpu} mesurent trois sortes de coûts pour la configuration étant donné un ensemble de composants, en supposant que les composants disponibles sont décrits par des contrats concernant les besoins en CPU, mémoire et énergie [7].

Des exemples de fonctions de caractérisation pour des propriétés de configuration sont données par les formules suivantes.

$$CF_{relevance}(context) = \begin{cases} 100 & \text{, si } networkBandwidth \geq 56 \\ 50 & \text{, si } 0 < networkBandwidth < 56 \\ 0 & \text{, si } networkBandwidth = 0 \end{cases} \tag{4.1}$$

$$CF_{memory}(components, context) = \begin{cases} 0 & \text{, si } \frac{\sum_{c \in components} maxCost_{memory}(c)}{value_{memory}} \leq 1 \\ 100 & \text{, sinon} \end{cases} \tag{4.2}$$

Dans la formule 4.1, la pertinence d'une configuration dépend exclusivement de la bande passante d'un réseau. La formule 4.2 donne un exemple de fonction de caractérisation pour la propriété de coût « mémoire ». Elle se base sur le ratio entre (a) le coût mémoire maximum d'un assemblage basé sur la configuration étant donné un ensemble de composants, et (b) le niveau courant de mémoire libre ($value_{memory}$). La valeur de la quantité maximale de mémoire requise $maxCost_{memory}$ est supposée fournie par chaque composant.

7. Si le domaine d'application ne requiert pas ce genre de préoccupations, alors les décisions ne se feront qu'en fonction de la pertinence et les composants n'auront pas à fournir d'informations sur leurs coûts.

4.2.5 Politique d'assemblage

```
"----- Déclenchement -----"
minDurationBetweenAdaptationTriggering := 2000.
contextualSituations := {
   (value_CPU > 1000) & (value_memory ≥ 512).
   (value_CPU > 1500) & (value_energy ≤ 10).
   }.
"------ Décision ------"
maxDurationForAdaptationDecision := 1000.
SF_bcc(config, components, context) := 40 * config.CF_relevance(context)
                                       −20 * config.CF_energy(components, context)
                                       −10 * config.CF_memory(components, context)
                                       −10 * config.CF_CPU(components, context).
```

FIGURE 4.5 – Exemple de spécification de politique d'assemblage.

Une politique d'assemblage dirige les (ré)assemblages. Elle spécifie quand (*i.e.* dans quel contexte) et comment (*i.e.* selon quels critères de décision) effectuer les assemblages. Dans MADCAR, la **politique d'assemblage** comporte deux parties utilisées respectivement pour (1) la détection des situations contextuelles qui peuvent nécessiter un (ré)assemblage, et (2) les règles qui guident les choix effectués par le moteur lors des réassemblages.

4.2.5.1 Déclenchement

La première partie de la politique d'assemblage concerne les conditions requises pour *déclencher une adaptation*. Elle consiste dans la spécification de :

- `contextualSituations` : une disjonction de plusieurs situations contextuelles, et
- `minDurationBetweenAdaptationTriggering` : la période minimale de temps entre deux évaluations des situations contextuelles données.

Une **situation contextuelle** est une conjonction de conditions portant sur des *données contextuelles*, c'est-à-dire des informations recueillies par les sondes du gestionnaire de contexte. Une adaptation est susceptible de survenir à chaque fois que le contexte courant correspond à une des situations contextuelles. Dans la politique d'assemblage de la figure 4.5, l'ensemble `contextualSituations` contient deux situations contextuelles. La première serait satisfaite par le contexte $\{value_{CPU} = 2200; value_{memory} = 1024; value_{energy} = 60\}$, mais pas la seconde.

La valeur de `minDurationBetweenAdaptationTriggering` limite l'utilisation de ressources par le moteur d'assemblage (à une durée de 2000 mil-

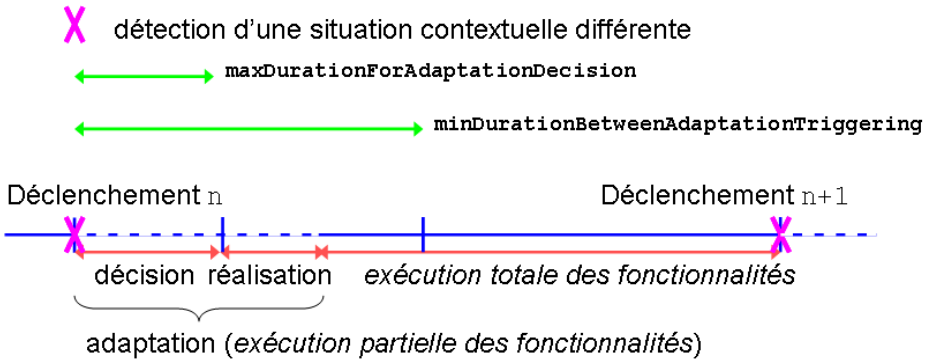

FIGURE 4.6 – Périodes d'adaptation et périodes d'exécution dans MaDcAr.

lisecondes dans l'exemple de la figure 4.5) et assure que chaque assemblage sera utilisé pendant une durée significative. Il est de la responsabilité du concepteur de l'application de spécifier une valeur assez grande, en fonction de la taille de l'application et des ressources matérielles disponibles. Le choix d'une valeur trop petite conduirait à pénaliser l'application voire à la figer totalement du fait d'adaptations trop fréquentes lorsque le contexte évolue rapidement. La figure 4.6 illustre les périodes d'adaptation et les périodes d'exécution dans MaDcAr. Le déclenchement correspond à une conjonction entre l'écoulement de `minDurationBetweenAdaptationTriggering` et la vérification d'une situation contextuelle.

4.2.5.2 Décision

La seconde partie de la politique d'assemblage consiste en la spécification de :

- SF_{bcc} : une fonction de sélection qui combine les fonctions de caractérisation d'une configuration afin de déterminer leurs impacts respectifs sur les décisions d'adaptation de l'application, et
- `maxDurationForAdaptationDecision` : la période maximale de temps allouée au processus de décision [8].

La sélection d'une meilleure configuration et d'un ensemble de composants consiste à maximiser la fonction de sélection SF_{bcc} fournie par le concepteur de l'application en fonction du contexte et des composants à choisir, jusqu'à ce que la durée `maxDurationForAdaptationDecision` ne s'achève [9].

8. Il est nécessaire que `maxDurationForAdaptationDecision` < `minDurationBetweenAdaptationTriggering`.

9. Bien que tentente, nous n'avons pas essayer d'obtenir un algorithme « *anytime* » [Zil96] pour le processus de réassemblage en supprimant le paramètre de temps

Un exemple de politique d'assemblage est donné par la figure 4.5. Cette politique montre une fonction de sélection définie par une simple fonction additive pondérée, mais le concepteur peut spécifier une fonction plus sophistiquée pourvu que les paramètres d'entrées soient une configuration, l'ensemble des composants et un contexte d'exécution.

4.2.6 Processus de (ré)assemblage

MADCAR peut être utilisé pour construire automatiquement des assemblages à partir de composants déconnectés et pour adapter (réassembler) dynamiquement et automatiquement des assemblages existants. Le processus de (ré)assemblage consiste en trois étapes successives : déclenchement, décision et réalisation.

1. *Déclenchement :* Le moteur d'assemblage vérifie périodiquement si le contexte courant correspond à une des situations contextuelles spécifiées dans la partie déclenchement de la politique d'assemblage (voir la figure 4.5).
2. *Choix d'un assemblage pertinent :* Le moteur d'assemblage sélectionne la configuration et l'ensemble de composants qui maximisent la fonction de sélection SF_{bcc} spécifiée dans la politique d'assemblage. Cette maximisation [LP01] passe par un problème de satisfaction de contraintes (CSP) avec :
   - une variable $V_{GoodConfig}$ dont le domaine est l'ensemble des configurations disponibles ;
   - un ensemble de variables V_{R_i} pour chaque rôle $R_i(min_i, max_i)$ des configurations avec pour domaine l'ensemble des composants disponibles et pour type une collection de taille comprise entre min_i et max_i ;
   - des contraintes sur chaque variable V_{R_i} qui sont les contrats du rôle R_i, auxquels on ajoute une contrainte qui assure que tous les composants sélectionnés pour les rôles doivent être distincts ;
   - une fonction objectif qu'il faut maximiser selon la politique d'assemblage : SF_{bcc}.

alloué à la décision. Ce processus de maximisation se focalise essentiellement sur le coût de la décision du réassemblage plutôt que sur le coût global du réassemblage (décision et réalisation). Le coût de la réalisation du réassemblage est tout à fait acceptable car il est nécessairement borné et probablement inférieur au coût de la décision. De plus, des problèmes de cohérences ou de performance pourraient survenir dans le cas d'un processus d'optimisation qui serait trop exigeant ou rigide sur le temps de réassemblage.

Cet unique CSP permet au moteur de sélectionner une des configurations et de sélectionner les composants à utiliser [10]. Parmi les configurations qui sont éligibles et pertinentes pour le contexte (c'est-à-dire où `relevance` est non nulle), le moteur doit choisir celle qui satisfait le mieux les préférences du concepteur.

3. *(Ré)assemblage :* Les composants actifs (associés à des processus légers) qui appartiennent à l'assemblage courant sont désactivés [11]. Ensuite, les nouveaux composants sélectionnés sont assemblés et initialisés selon la configuration choisie. Finalement, les composants assemblés sont activés. Pendant cette étape, l'état de l'application est maintenu à travers un mécanisme de transfert d'état (voir la section 4.3).

MADCAR traite aussi bien les réassemblages profonds que les réassemblages légers. Un **réassemblage profond** survient uniquement lorsque la configuration cible et la configuration source sont différentes. Dans ce cas, la structure de l'assemblage courant est modifiée à cause du changement de configuration. Un **réassemblage léger** survient uniquement lorsque la configuration cible et la configuration source sont les mêmes. Dans ce cas, la structure de l'assemblage courant est stable, mais certains composants peuvent être remplacés. De même, des composants peuvent être ajoutés ou supprimés sur la base des multiplicités des rôles impliqués et la politique d'assemblage.

4.2.7 Gestion de la dynamicité de l'adaptation

Le processus de réassemblage peut survenir pendant l'exécution de l'application. Il faut adapter l'application *dynamiquement*. Par conséquent, le moteur d'assemblage doit assurer la *cohérence* de l'application en gérant non seulement l'état de l'application mais aussi l'activité de l'application. La gestion de l'état est explicitée en section 4.3.

10. Concrètement, cette étape de décision par optimisation de CSP comporte deux phases :

(a) phase initiale : obtenir une solution d'assemblage satisfiante en interogeant le solveur de contraintes (sans parcourir tout l'espace de recherche),

(b) phase récurrente : tant que `maxDurationForAdaptationDecisions` n'est pas écoulée, alors obtenir une solution meilleure vis-à-vis de la fonction objectif (en continuant le parcours de l'espace de recherche).

11. Cela présuppose qu'ils soient dotés d'une interface extra-fonctionnelle permettant de contrôler leur activité. La désactivation/réactivation d'un ensemble de composants est reprise des travaux existants [KM90, BHA+05].

Les composants actifs de l'application - lorsqu'il y en a - peuvent être désactivés ou réactivés au besoin par le moteur d'assemblage. Cela signifie que l'activité de ces composants, c'est-à-dire leur comportement (thread [12] interne, interactions en cours) est contrôlable grâce à une interface extra-fonctionnelle. Les composants se désactivent proprement dans un délai « raisonnable » prévu par leurs concepteurs grâce à cette interface d'activité, en arrêtant leurs threads internes ainsi que leur communication courante. Les mécanismes et l'ordre utilisés pour la désactivation/réactivation d'un ensemble de composants sont issus des travaux tels que [KM90, Ajm04, BHA+05]. Une application est dite *stoppée* uniquement lorsque tous ses composants actifs sont désactivés.

La gestion de la dynamicité diffère entre un réassemblage léger et un réassemblage profond. Dans le cas d'un réassemblage profond, l'exécution de tout l'assemblage de composants est stoppée pendant l'adaptation de l'application. En d'autres termes, l'adaptation dynamique - et a fortiori la continuité de services de l'application - n'est pas considérée dans le cas d'un réassemblage profond. Après avoir stoppé l'application, il suffit de construire un nouvel assemblage, comme le préconise Medvidovic [Med96].

Dans le cas d'un réassemblage léger, le moteur d'assemblage s'efforce de minimiser l'impact de l'adaptation sur l'assemblage courant. En effet, l'exécution de l'assemblage n'est pas stoppée mais seuls les composants directement affectés par un changement et les composants qui leurs sont directement adjacents (*i.e.* connectés par une interface fourni ou requise avec des composants directement affectés) sont désactivés. Ceci assure que les composants directement affectés par une adaptation ne recevront pas de requêtes : l'application est dite dans un « état quiescent » [KM90]. De plus, pendant l'adaptation d'une partie de l'assemblage, des queues sont utilisées par le moteur d'assemblage pour stocker temporairement les requêtes envoyées aux composants désactivés de la part des autres composants. A la fin du réassemblage, tous les composants sont réactivés et toutes les requêtes en attente sont renvoyées aux composants du nouvel assemblage.

4.2.8 Modèle d'application auto-adaptable

L'une des utilisations les plus pertinentes des infrastructures d'adaptation est de permettre des logiciels auto-adaptables, selon les principes de l'informatique autonomique (ou « *Autonomic Computing* » [KC03]). Le modèle MADCAR ne déroge pas à cette possibilité puisqu'il suffit de définir notre

12. Processus léger.

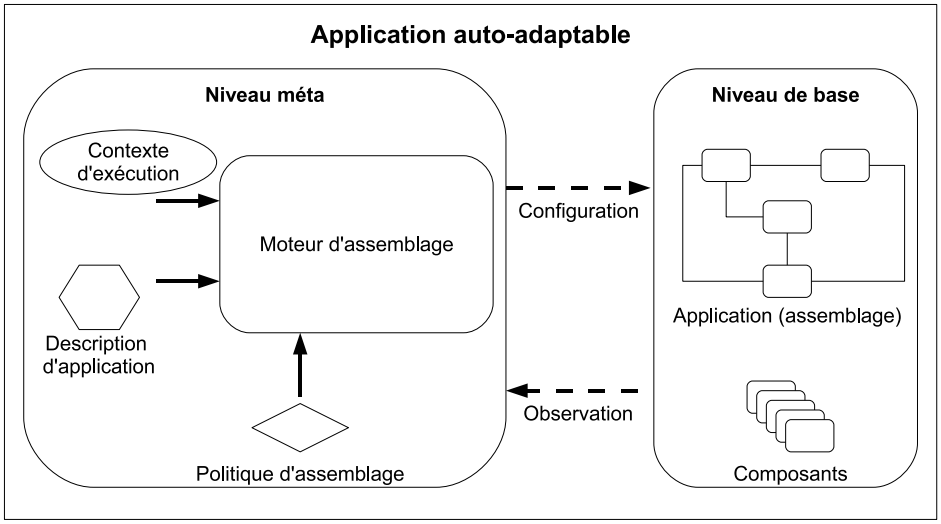

FIGURE 4.7 – Modèle d'application auto-adaptable.

infrastructure d'adaptation comme une composante interne à l'application.

La figure 4.7 montre un modèle d'application auto-adaptable fondé sur le concept de moteur d'assemblage. Nous proposons de définir une couche méta qui contrôle les réassemblages automatiques des composants constituant le niveau de base. La description d'application décrit alors les assemblages valides pour le niveau de base. Ainsi, le fait de construire une application auto-adaptable à base de composants avec MADCAR se résume à considérer l'infrastructure d'adaptation (notamment le moteur d'assemblage) comme faisant partie intégrante de l'application. Sur ce principe, nous avons utilisé MADCAR pour construire des agents logiciels qui s'adaptent de manière autonome (voir le chapitre 5).

4.3 Gestion de l'état d'une application lors des adaptations

La reconfiguration « à chaud » pose le classique, mais néanmoins épineux, problème de *transfert d'état* [SF93]. En effet, l'état de l'application doit rester cohérent quand il y a remplacement de composants. Nous décrivons ici un formalisme qui permet au concepteur de l'application de définir l'état de l'application et de spécifier les règles de transfert d'état de manière abstraite, indépendamment des composants logiciels. Un des problèmes auxquels nous devons faire face est que l'état ne peut être stocké dans les composants. Du fait du découplage entre les configurations et les composants,

un même composant peut être utilisé pour remplir des rôles différents au fil des réassemblages.

4.3.1 Etat d'une application

L'état d'une application dans MADCAR s'appuie sur les attributs définis dans les rôles de la configuration (voir la sous-section 4.2.4). Nous définissons l'**état d'une application** comme l'ensemble des valeurs de tous les attributs variables spécifiés dans les configurations. Les valeurs courantes de ces attributs correspondent à l'état courant de l'application.

Lorsqu'un composant qui satisfait un rôle R est supprimé d'un assemblage, les attributs des composants qui sont marqués comme variables dans R doivent être sauvegardés. Ces données sont restaurées dans n'importe quel autre composant qui sera choisi pour remplir R par la suite. Notons que dans le cas des attributs d'un rôle ayant une multiplicité supérieure à un, c'est un vecteur de valeurs qui est sauvegardé pour mémoriser les attributs des composants jouant ce rôle. Un vecteur de taille n peut initialiser au plus n composants et s'il reste des composants non initialisés, alors on utilise une *fonction d'initialisation* définie dans chaque rôle.

4.3.2 Principe du transfert d'état

Dans une application, les attributs variables de deux configurations peuvent être liés. Il y a liaison entre deux attributs variables si la modification de la valeur de l'un doit être répercutée sur la valeur de l'autre pour que l'état de l'application reste cohérent. De telles liaisons relèvent de la sémantique de l'application et sont donc difficiles à déterminer de manière automatique. C'est donc au concepteur de l'application que revient la responsabilité de les spécifier. Pour ce faire, le transfert d'état doit être spécifié sous la forme d'un *réseau de transfert d'état* qui explicite les relations entre les attributs des configurations.

Le principal intérêt du réseau de transfert d'état est qu'il permet de déduire une séquence valide de transferts et de conversions pour la mise à jour cohérente des valeurs des attributs. C'est-à-dire une séquence qui préserve la cohérence de l'état de l'application et évite la perte d'information. En effet, certaines configurations utilisent des données qui ne sont pas utilisées dans d'autres configurations de l'application et des données qui sont présentes dans toutes les autres configurations, par exemple lorsqu'une configuration contient un rôle complètement différent des autres rôles des configurations. C'est pourquoi il faut maintenir les transferts d'état de manière globale dans un réseau.

D'après [VB03], on peut identifier deux grandes approches pour représenter l'état d'une application à base de composants : *l'approche basée sur un modèle de représentation* et *l'approche basée sur l'implémentation*. La première approche consiste à utiliser une entité « pivot » entre les composants qui sont interchangeables. Dans la deuxième approche, l'importation de l'état d'un composant doit être basée sur l'implémentation de l'exportation du composant à remplacer. Notre proposition pour MADCAR peut être vue comme une généralisation de la première approche. En effet, les rôles appartenant aux configurations jouent le rôle de pivots envers les composants, mais surtout le réseau de transfert d'état permet de spécifier les dépendances entre ces différents pivots. Par ailleurs, il faut noter que notre travail porte sur le transfert d'état de toute l'application dans le cas d'un changement potentiellement profond de l'architecture et ne se restreint pas au seul cas d'un remplacement de composant.

4.3.3 Spécification des règles de transfert d'état

Le transfert d'état doit être spécifié sous la forme d'un **réseau de transfert d'état**.

- Les nœuds de ce réseau sont essentiellement les attributs variables et parfois des *nœuds virtuels* (voir la sous-section 4.3.4).
- Les liens connectent uniquement des attributs définis dans des configurations différentes et sont orientés. On distingue trois types de liens : un *lien simple* connecte deux nœuds, un **lien join** connecte un groupe de nœuds vers un seul nœud et un **lien fork** connecte un nœud vers un groupe de nœuds.
- Chaque lien porte exactement deux *fonctions de transfert* pour propager des valeurs dans les deux sens du lien.

Ces différentes notions sont explicitées par la suite.

4.3.3.1 Liens simples

Les **fonctions de transfert** permettent de calculer les valeurs d'un attribut à partir d'un autre. Étant données deux nœuds a et b reliés par un lien, le concepteur doit fournir :

- $Transfer_{a-b}$ qui permet de calculer la valeur de b à partir de celle de a, et
- $Transfer_{b-a}$ qui permet de calculer la valeur de a à partir de celle de b.

L'orientation des liens dénote les risques de pertes d'informations. Considérons par exemple un lien orienté de a vers b (noté $a \triangleright b$). Cette orientation

montre que a est *moins riche que* b, c'est-à-dire qu'il n'y a pas de perte d'information lorsqu'on transfère la valeur de b vers a et qu'en revanche il peut y avoir une perte d'information lorsqu'on transfère la valeur de a vers b. Pour éviter cette perte, la fonction de transfert $Transfer_{b-a}$ peut utiliser l'ancienne valeur de b pour calculer la nouvelle valeur.

Lorsqu'il y a un lien $a \triangleright b$, nous disons que b est **maître** de a et que a est **esclave** de b. Cette hiérarchie est transitive. Ainsi, si c est maître de b, alors c'est un maître indirect de a. Ainsi, la valeur de a peut être calculée à partir de c en passant par b et inversement. Nous parlons de *propagation* de valeurs. Un attribut maître qui n'a pas de maître est appelé un **maître absolu**. L'état d'une application peut être restreint à l'ensemble des maîtres absolus, puisque la valeur de tout autre d'attribut peut être déduite - par propagation - à partir des attributs maîtres absolus.

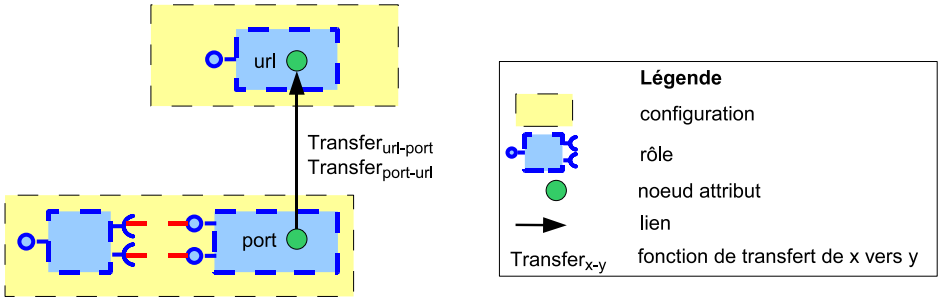

FIGURE 4.8 – Exemple simple de réseau de transfert d'état dans MADCAR.

Pour illustration, considérons deux configurations dont la première contient un rôle avec un attribut variable *url* qui représente l'adresse internet d'un serveur et dont la seconde inclut un attribut variable *port*. Pour cet exemple, *url* est une chaîne de caractères composée de deux parties, un nom de serveur et le port du serveur, qui sont séparées par un caractère ' :'. Une illustration du réseau de transfert d'état correspondant est montrée dans la figure 4.8. Les fonctions de transfert sont les suivantes :
$Transfer_{port-url}$: `url = url.cutBeforeLast(' :') + ' :' + port.toString()`.
$Transfer_{url-port}$: `port = url.cutAfterLast(' :').toInteger()`.

4.3.3.2 Liens multiples

Un réseau de transfert d'état peut comporter des *liens multiples* car une information dans une configuration peut être séparée en plusieurs parties dans d'autres configurations.

De la même manière que pour les liens simples, les liens multiples portent des fonctions de transfert. Par exemple, un lien entre un nœud a et un groupe de nœuds $G_B = \{b_1, ..., b_j\}(j > 1)$ doit être « marqué » par deux fonctions de transfert : (1) $Transfer_{a-b_1,b_2,...,b_j}$ permettant de calculer les valeurs des nœuds du groupe G_B à partir de a et (2) $Transfer_{b_1,b_2,...,b_j-a}$ qui calcule a à partir de G_B.

Par ailleurs, les liens multiples sont également orientés. L'ensemble des nœuds cibles d'un lien *join* est appelé **groupe maître**, et l'ensemble des nœuds sources d'un lien *fork* est appelé **groupe esclave**.

4.3.4 Règles de cohérence

Afin de garantir la cohérence pendant les transferts d'état, les réseaux de transfert d'état doivent obéir aux règles suivantes :

1. chaque attribut peut avoir au plus un maître direct : soit un attribut maître simple, soit un groupe d'attributs maîtres ;
2. chaque attribut peut avoir au plus un esclave direct : soit un attribut esclave simple, soit un groupe d'attributs esclaves ;
3. la connexion entre deux groupes de nœuds, passe obligatoirement par un **nœud virtuel**, c'est-à-dire un nœud qui ne correspond à aucun attribut dans les rôles des configurations ;
4. les cycles sont interdits ;
5. si des attributs $\{a_1, ..., a_j\}(j > 1)$ qui n'ont pas de maître sont sémantiquement liés (*i.e.* ont des informations en commun) et si aucun d'eux n'est moins riche que les autres, alors un *nœud virtuel* v doit être ajouté au réseau tel que $\{a_1, ..., a_j\} \triangleright v$ (*i.e.* lien *join* du groupe $\{a_1, ..., a_j\}$ vers un maître v).

Ces règles assurent qu'il est toujours possible de définir un ensemble d'attributs maîtres absolus [13]. De plus, elles assurent que le chemin de transfert entre deux attributs - lorsqu'il y en a un - est unique puisque à partir de n'importe quel nœud, une seule fonction de transfert peut être utilisée pour propager une valeur vers d'autres nœuds (vers le haut ou vers le bas).

4.3.5 Transfert d'état

Le transfert d'état survient pendant le réassemblage (sous section 4.2.6). Il est réalisé en quatre phases :

13. Au pire, il suffit d'utiliser la dernière règle pour créer un nœud virtuel qui serait l'unique maître absolu en agrégeant les valeurs de tous les autres attributs.

1. lecture depuis les composants de l'assemblage courant des valeurs correspondant aux attributs variables pour chaque rôle de la configuration courante ;
2. propagation des valeurs des attributs variables de la configuration courante vers les attributs maîtres absolus de l'application en se basant sur le réseau de transfert d'état ;
3. propagation des valeurs des attributs variables des attributs maîtres absolus vers tous [14] les attributs (notamment ceux de la nouvelle configuration) en se basant sur le réseau de transfert d'état ;
4. initialisation des attributs des composants qui ont été sélectionnés pour la nouvelle configuration : les valeurs des attributs variables proviennent du réseau de transfert d'état tandis que les valeurs des attributs fixes sont directement issues de leur spécification dans les rôles.

Le processus décrit ci-dessus s'opère quand il y a passage d'une configuration à une autre. Un cas particulier correspond au démarrage de l'application. Dans ce cas, aucune valeur n'est disponible pour les attributs fixes et variables de la configuration de démarrage. C'est pourquoi nous utilisons des fonctions d'initialisation. Chaque rôle contient une fonction d'initialisation pour ses attributs fixes et variables (voir la sous-section 4.2.4). De plus, les nœuds virtuels ont aussi leurs propres fonctions d'initialisation lorsqu'ils sont des maîtres absolus.

Un exemple de réseau de transfert d'état est montré dans la figure 4.9. Les attributs maîtres absolus sont v (qui est un nœud virtuel), b, c, d, e et h. Considérons l'initialisation de l'application sachant que la configuration choisie est $C3$. Elle se déroule en trois temps. Premièrement, les valeurs des maîtres absolus (virtuels ou non) sont calculées à l'aide des fonctions d'initialisation puis propagées vers tous les autres nœuds du réseau de transfert d'état. Deuxièmement, il s'agit de compléter l'initialisation de la configuration $C3$. Pour chaque attribut variable des rôles de $C3$ qui ont une multiplicité maximale strictement supérieure à un, la fonction d'initialisation du rôle correspondant est utilisée pour que le vecteur de valeurs de cet attribut ait la même taille que le nombre de composants qui sont sélectionnés pour jouer ce rôle. Une fois l'initialisation de l'application passée, tous les attributs fixes et variables ont une valeur.

14. On propage systématiquement vers tous les attributs pour être certain qu'ils soient à jour lors du prochain changement de configuration, pour la propagation de l'étape 2. Par exemple, dans la figure 4.9, la propagation de la valeur de a vers le maître absolu v pendant l'étape 2 grâce au lien *join* $\{a, f\} \triangleright v$ nécessite l'utilisation de la valeur de f, c'est pourquoi il faut avoir propagé la valeur de v vers f en même temps que vers a pendant l'étape 3 du réassemblage précédent.

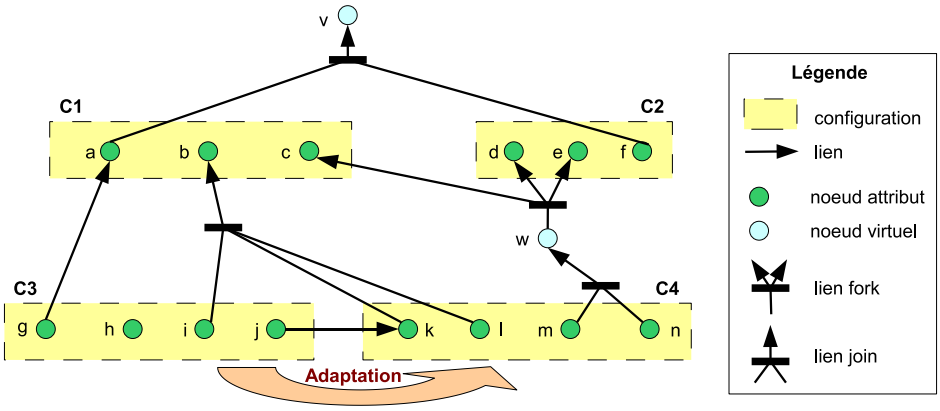

FIGURE 4.9 – Exemple élaboré de réseau de transfert d'état dans MADCAR.

Considérons à présent une adaptation de la configuration $C3$ vers la configuration $C4$. La *première phase* du transfert d'état consiste à lire les valeurs de g, h, i et j depuis les composants qui remplissent les rôles de $C3$. Dans la *deuxième phase*, les valeurs des attributs de $C3$ sont propagées successivement jusqu'à un maître absolu et dans un ordre arbitraire (disons g, h, i, j). La valeur de g est propagée jusqu'à l'attribut a de $C1$ puis la valeur de a est propagée vers le maître absolu v grâce au lien *join* $\{a, f\} \triangleright v$ en utilisant la valeur temporaire de f et en appliquant $Transfer_{a,f-v}$. L'attribut h est un maître absolu et donc ne nécessite pas de propagation. La valeur de i est propagée vers b en appliquant $Transfer_{i,k,l-b}$, la valeur de j est propagée vers l'attribut k de $C4$ et la valeur de k est à son tour propagée vers b en appliquant $Transfer_{i,k,l-b}$ pour la deuxième fois [15]. La *troisième phase*, qui a lieu à la fin de la construction du nouvel assemblage, consiste à propager les valeurs des attributs maîtres absolus jusqu'aux attributs de l'ensemble des configurations. Par exemple, la valeur de b est propagée vers le bas pour calculer les valeurs de i, k et l. Dans la *quatrième phase*, chaque valeur des attributs de la configuration $C4$ est utilisée pour initialiser les composants du nouvel assemblage.

15. La première fois que la fonction $Transfer_{i,k,l-b}$ est appliquée, b est calculé à partir de la nouvelle valeur de i et des valeurs non modifiées de k et l. La deuxième fois, les nouvelles valeurs de i et k sont toutes les deux utilisées pour recalculer b. Comme l n'a pas été modifié, le calcul de b est terminé.

4.4 Évaluation et comparaison avec l'état de l'art

4.4.1 Degré d'adaptabilité de MaDcAr

Cohérence	Performance	Disponibilité	Simultanéité	Ouverture
+++	+++	++	+++	+++

Table 4.1 – Critères de l'adaptation pour MaDcAr

Ci dessous, nous décrivons le modèle MaDcAr selon les critères définis en 2.3.8. Le tableau 4.1 synthétise l'évaluation de MaDcAr par rapport à ces critères d'adaptation.

Cohérence : le besoin de cohérence est complètement satisfait car les opérations simples de réassemblage (connexion, déconnexion, etc.) sont supposées être sûres étant donné un modèle de composant et que les opérations complexes (basées sur un ensemble de configurations supposées valides) sont automatiquement décomposées en opérations simples grâce issues d'une solution fournie par le solveur de contraintes de MaDcAr. De plus, les problèmes de transfert d'état dans le cas dynamique sont traités.

Performance : le besoin de performance pendant l'adaptation est spécifiquement pris en compte dans la politique d'adaptation et permet au concepteur de définir un compromis entre la quantité de ressources matérielles utilisées par le processus d'adaptation et la durée des adaptations. La durée des adaptations est bornée d'une part parce que le temps alloué à la décision est fixé par le concepteur, et d'autre part, même lorsque le temps alloué à la décision expire, la décision d'assemblage choisie par défaut est la moins mauvaise (parmi les possibilités déjà évaluées) et comme cette décision porte sur des configurations établies par le concepteur de l'application, il est certain que le moteur d'assemblage pourra la réaliser assez rapidement.

Disponibilité : le besoin de disponibilité des services offerts par l'application pendant l'adaptation est partiellement pris en compte grâce au mécanisme de transfert d'état et de temporisation des appels vers le composant parent du composant concernée par l'adaptation. Cependant, le nombre de composants bloqués inutilement peut être grand.

Simultanéité : le cas des adaptations simultanées est complètement pris en compte car une opération d'adaptation consiste à adopter une des

configuration prévues selon la politique d'adaptation. Chaque opération d'adaptation peut concerner plusieurs composants, voire même remplacer l'ensemble des composants de l'application. Ces adaptations complexes consistent à composer différentes opérations simples (ajout et suppression de composants).

Ouverture : le degré d'ouverture de notre infrastructure d'adaptation est important car le processus d'adaptation de MADCAR est configurable à travers une politique d'assemblage qui spécifie le contexte, le moment et les conditions de déclenchement des adaptations, ainsi que les composants concernés par l'opération de réassemblage et l'ordonnancement des étapes des réassemblages. Typiquement, dans le cas d'une situation contextuelle indiquant une faible quantité des ressources matérielles disponibles, l'administrateur de l'application doit exprimer dans la politique d'assemblage un compromis stratégique entre (a) le fait d'allonger la durée d'adaptation pour favoriser le choix des adaptations optimales (*i.e.* celles qui maximisent la fonction de sélection) et (b) le fait de limiter la durée d'adaptation pour les services applicatifs ne manquent pas de ressources matérielles, quitte à effectuer des adaptations qui ne satisfont que partiellement les préférences d'adaptation (pertinence, CPU, mémoire et énergie).

4.4.2 Comparaison de MADCAR avec l'état de l'art

Dans le chapitre 2, nous avons étudié plusieurs travaux proposant des infrastructures pour l'adaptation d'applications à base de composants, notamment SAFRAN, SOFA et CASA. Contrairement à MADCAR, ces travaux sont spécifiques à un modèle de composants particulier. Le tableau 4.2 récapitule les évaluations de MADCAR et d'autres infrastructures d'adaptation. Dans ces travaux, le processus d'adaptation dynamique et automatique est entièrement contrôlé par le concepteur de l'application. De plus, la spécification des adaptations est exprimée à un haut niveau (grâce à une politique d'adaptation) dans le cas de MADCAR, SAFRAN et CASA.

La spécification des adaptations dans MADCAR est à la fois globale à l'application (comme pour CASA) et découplée des composants. Dans SAFRAN et SOFA, elle est locale aux composants ou aux connecteurs, ce qui peut entraîner des problèmes d'incohérence ou dans le cas de SAFRAN un surcoût de performance dû à la correction a posteriori des adaptations incohérentes. MADCAR permet au concepteur de l'application d'avoir le contrôle sur la performance des adaptations car il fixe le temps alloué à la décision des adaptations.

	Cohérence	**Performance**	**Disponibilité**	**Simultanéité**	**Ouverture**
Travaux dans le domaine des langages de description d'architecture					
C2	++	+	++	++	++
Darwin	++	+	++	++	++
Travaux liés à un modèle de composants					
SAFRAN	+++	+	++	++	+++
Think	++	++	++	++	++
SOFA	++	+	++	++	++
CASA	+++	++	+++	+++	+++
Notre proposition					
MADCAR	+++	+++	++	+++	+++

TABLE 4.2 – Récapitulatif des ABCs adaptables (dont MADCAR)

Concernant le transfert d'état lors des adaptations, MADCAR et CASA adoptent l'approche basée sur un modèle de représentation. Par contre, SAFRAN et SOFA utilisent l'approche (*ad hoc*) basée sur l'implémentation (voir la sous-section 4.3.2).

Enfin, seul MADCAR offre l'avantage de pouvoir réutiliser les spécifications des adaptations définies par un concepteur avec des composants différents. En effet, l'élément principal constituant une politique d'adaptation est la fonction de sélection qui permet de choisir une configuration et des composants à assembler, en fonction de critères objectifs (pertinence au contexte applicatif et aux ressources matérielles disponibles) et dans des proportions définies par le concepteur. En outre, la notion de configuration dans MADCAR est sensiblement différente que dans les autres travaux, notamment les langages de description d'architecture. Nos configurations ne font pas directement référence à des composants mais à des descriptions fonctionnelles et extra-fonctionnelles que nous appelons des rôles et qui sont moins rigide que des types. Ainsi, une même configuration peut être réutilisée avec d'autres composants.

Finalement, notons que l'utilisation d'un solveur de contraintes dans le processus de décision des adaptations marque la volonté de permettre au concepteur de spécifier les adaptations de manière déclarative mais sans avoir à raisonner au cas par cas sur les possibles situations contextuelles.

4.5 Conclusion

Dans ce chapitre, nous avons présenté MADCAR, un modèle de moteurs capable d'assembler et d'adapter automatiquement et dynamiquement des applications à base de composants. Dans MADCAR, un moteur d'assemblage possède quatre entrées : un ensemble de composants à assembler, une description de l'application qui représente la spécification de l'assemblage en termes de propriétés fonctionnelles et extra-fonctionnelles, une politique

d'assemblage qui dirige les décisions d'adaptation et un contexte qui contient un ensemble de données (état de l'application, CPU disponible, bande passante, etc.) mesurées par des sondes. Au démarrage d'une application, le moteur MadCar détermine une première configuration (description d'un assemblage) et un ensemble de composants à assembler pour construire l'application. Lorsque le contexte change, le moteur choisit une configuration plus appropriée et réassemble les composants disponibles en conséquence. Ainsi, le même mécanisme d'assemblage automatique s'applique à la fois à la construction des applications et à leur adaptation.

Une caractéristique majeure de MadCar est qu'il permet aux concepteurs de produire des spécifications génériques. La spécification de l'architecture d'une application et de ses adaptations sont totalement découplées des composants logiciels. Aucune référence directe sur les composants n'est admise. Par ailleurs, la politique d'assemblage est séparée de la description de l'application. Ainsi, MadCar encourage la séparation des préoccupations.

Par ailleurs, MadCar prend en charge les adaptations non anticipées dans la mesure où les descriptions d'application et les composants peuvent être changés pendant l'exécution, c'est-à-dire sans stopper toute l'application. La spécification des adaptations est à la fois globale à l'application et découplée des composants. De ce fait, l'approche MadCar présente l'avantage d'éviter les problèmes de cohérence des adaptations rencontrés dans les approches où chaque composant s'adapte indépendamment des autres. Enfin, les spécifications des adaptations peuvent être réutilisées avec des composants différents. Enfin, comme notre approche d'adaptation est indépendante d'un modèle de composants particulier, nous proposons un formalisme qui permet au concepteur de l'application de définir l'état de l'application et de spécifier les règles de transfert d'état de manière générique.

Pour finir, notons que le fait de construire une application auto-adaptable à base de composants avec MadCar se résume à considérer l'infrastructure d'adaptation (notamment le moteur d'assemblage) comme une composante même de l'application. Sur ce principe, nous avons utilisé MadCar pour construire des agents logiciels qui s'adaptent de manière autonome (voir le chapitre suivant).

Chapitre 5

MaDcAr-Agent : un modèle d'agents auto-adaptables

MaDcAr-Agent est un modèle abstrait d'agent à base de composants. Ce modèle décrit la structure d'une architecture d'agent auto-adaptable qui s'appuie sur le moteur de réassemblage MaDcAr, décrit dans le chapitre précédent. Cette architecture permet d'assembler automatiquement et dynamiquement les composants internes d'un agent. Reposant sur le modèle MaDcAr pour les configurations, MaDcAr-Agent est indépendant de tout modèle spécifique de composant logiciel.

MaDcAr-Agent enrichit MaDcAr en étendant les possibilités de l'adaptation d'après les capacités propres à un agent logiciel évoluant dans un système multi-agent (SMA). Selon la définition proposée par Wooldridge et Jennings,

> « un **agent** est un système informatique matériel ou (plus souvent) logiciel qui bénéficie des propriétés suivantes :
> – autonomie : les agents agissent sans l'intervention directe d'humains ou d'autres intervenants, et ont un certain contrôle sur leurs actions et états internes ;
> – capacité sociale : les agents interagissent avec d'autres agents (et éventuellement des humains) à l'aide d'un langage de communication agent ;
> – réactivité : les agents perçoivent leur environnement [...] et répondent de manière opportune aux changements qui y surviennent ;
> – proactivité : les agents n'agissent pas seulement en réponse à leur environnement, ils sont capables de faire preuve d'un comportement dirigé par des buts en prenant des initiatives » [WJ95].

L'utilisation du moteur MaDcAr gérant l'adaptation est pertinente pour les agents car ce moteur de réassemblage leur permet de modifier leur comportement ainsi que leur structure en fonction des changements perçus dans l'environnement. En particulier, même si le modèle MaDcAr permet de concevoir des applications auto-adaptables (voir la section 4.2.8), les possibilités d'adaptation offertes par les spécifications du concepteur (configurations et politique d'assemblage) sont limitées par le nombre et la variété des composants fournis par le concepteur. Parfois, ce dernier ne peut pas (*e.g.* mission de l'application pouvant évoluer de façon imprévisible) ou ne veut pas (*e.g.* économie de ressources matérielles) inclure tous les composants utilisables par l'application. Dans le cadre d'agents, les adaptations doivent se faire de manière la plus autonome et pro-active possible, ainsi qu'en exploitant la dimension sociale. C'est pourquoi, il est intéressant de permettre à un agent de rechercher lorsqu'il le souhaite de nouveaux composants, en interagissant avec d'autres agents.

La première partie de ce chapitre donne une vue globale du modèle MaDcAr-Agent en décrivant les différents niveaux de son architecture (section 5.1). Ensuite, la section 5.2 décrit les mécanismes qui contrôlent la capacité d'un agent à gérer (ajout et suppression) les composants qu'il contient en fonction de leur utilité et des ressources disponibles, notamment en s'appuyant sur les échanges avec les autres agents. Puis, la section 5.3 illustre la possibilité d'utiliser les architectures classiques d'agents (réactifs, délibératifs, interagissants, etc.) et de passer automatiquement d'une architecture à une autre pour optimiser la consommation de ressources matérielles par l'agent. La section 5.4 propose une évaluation du modèle et une comparaison avec l'état de l'art. Enfin, la section 5.5 fournit un résumé de notre approche.

5.1 Architecture générale de MaDcAr-Agent

La figure 5.1 esquisse l'architecture générale du modèle MaDcAr-Agent. Cette architecture définit trois niveaux principaux dans un agent. Le *niveau infrastructure* est l'interface entre l'agent et son infrastructure de déploiement (par exemple, un robot ou un téléphone mobile). Le *niveau de base* est la « partie opérationnelle » de l'agent, c'est-à-dire l'ensemble de composants implémentant des comportements d'agents, des connaissances ou des compétences permettant de réaliser des tâches spécifiques, notamment celles liées à un domaine d'application. Le *niveau méta* est celui qui est en charge des adaptations. Il encapsule un moteur MaDcAr qui est connecté au niveau infrastructure et au niveau de base pour percevoir respectivement le

contexte externe et le contexte interne de l'agent. De plus, le niveau méta peut modifier le niveau de base lorsque c'est nécessaire. C'est précisément cette opération qu'on appelle adaptation d'un agent dans ce chapitre.

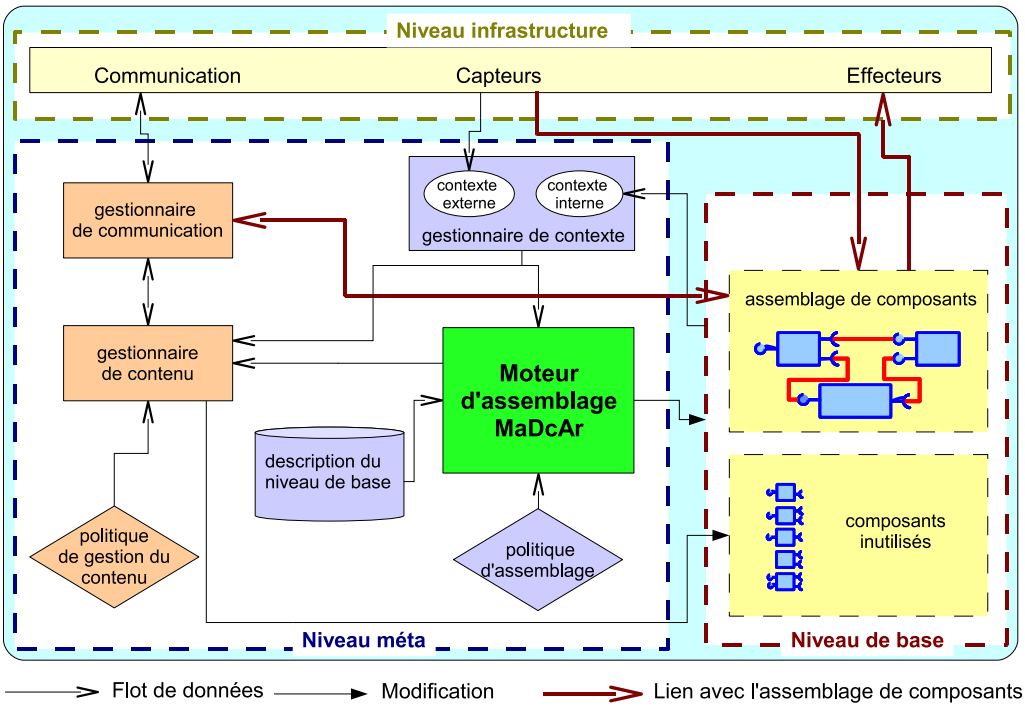

Figure 5.1 – Architecture générale de MaDcAr-Agent.

5.1.1 Niveau infrastructure

Un agent est toujours déployé sur une *infrastructure de déploiement*, c'est-à-dire une infrastructure matérielle et logicielle qui fournit des fonctionnalités pour percevoir et agir sur des entités extérieures. Par exemple, un intergiciel peut être utilisé comme une infrastructure de communication à l'intérieur d'un SMA. Ainsi, le niveau infrastructure d'une architecture d'agent est l'interface entre d'une part, les niveaux de base et méta et d'autre part, les fonctionnalités offertes par l'infrastructure.

5.1.1.1 Description du niveau infrastructure

Nous considérons que l'infrastructure est composée de trois parties :

– Les *capteurs* recueillent des informations sur le monde (physique ou virtuel) autour de l'agent. Les informations perçues sont ensuite exprimées dans une modélisation du « contexte externe » de l'agent. Le

modèle de contexte et son contenu ne sont pas détaillés ici puisqu'ils sont dépendants de l'application. Néanmoins, nous supposons qu'il y a au moins un capteur - nous l'intitulons `neighborhoodSensor` - qui permet de détecter le voisinage d'un agent. Son utilisation sera décrite par la suite. Nous définissons le **voisinage d'un agent** comme l'ensemble des agents avec lesquels il peut communiquer. Notons que les ressources de l'environnement ne sont pas des éléments du voisinage et qu'elles sont accédées par d'autres capteurs en fonction du type d'application.

– Les *effecteurs* permettent à un agent d'agir sur son environnement. Les actions possibles sont dépendantes de l'application et ne sont donc pas détaillées dans ce modèle générique. Par exemple, en robotique, une action possible peut être de faire avancer un robot ou bien de le faire changer de direction. Contrairement au cas des capteurs, aucun effecteur particulier n'est requis par notre approche.
– La partie *communication* est utilisée pour envoyer des messages vers d'autres agents et pour recevoir des messages de la part d'autres agents. Les messages peuvent être uniquement envoyés à des agents qui sont dans le voisinage de l'émetteur. La délimitation exacte du voisinage dépend de l'application ou de la technologie de communication utilisée. La partie communication est un service de bas niveau. Elle encapsule des fonctionnalités de communication de l'intergiciel.

5.1.1.2 Interactions avec les autres niveaux

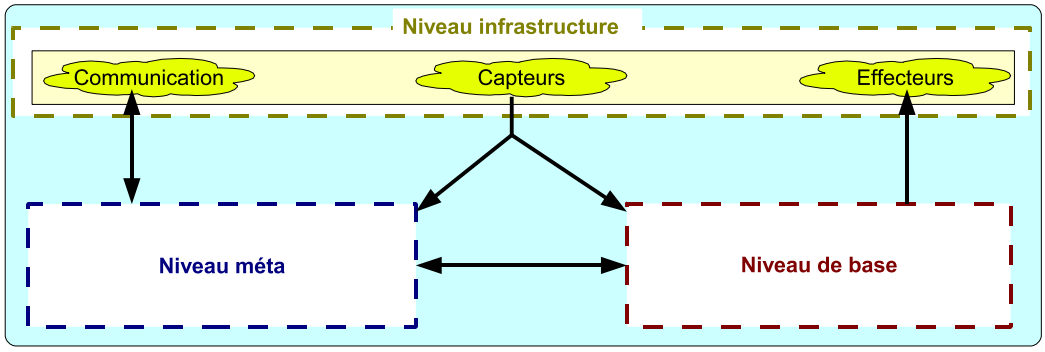

FIGURE 5.2 – Interactions du niveau infrastructure avec les autres niveaux.

La figure 5.2 illustre de manière schématique les interactions du niveau infrastructure avec les autres niveaux dans MADCAR-AGENT. La connexion entre le niveau infrastructure et le niveau de base dépend de l'assemblage de composants courant de l'agent. Les parties capteurs et effecteurs du niveau

infrastructure peuvent être utilisées par des composants du niveau de base lorsque c'est nécessaire, c'est-à-dire en fonction du comportement défini par l'assemblage de composants.

Le niveau méta interagit également avec le niveau infrastructure.

- Premièrement, les capteurs remplissent et mettent à jour une représentation du contexte externe qui est requis par le niveau méta. Par exemple, le capteur `neighborhoodSensor` peut être utilisé par le niveau méta pour rechercher des composants appartenant à des agents du voisinage (voir la sous-section 5.2.3).
- Deuxièmement, le niveau méta prend en compte les éléments du niveau infrastructure pendant une adaptation afin de modifier les connexions entre des composants du niveau de base et le niveau infrastructure. En particulier, si le niveau méta remplace un composant de l'assemblage courant qui était relié au niveau infrastructure (par exemple, un composant contenant la séquence d'actions à réaliser par les effecteurs), alors il faut aussi connecter le nouveau composant au niveau infrastructure.
- Finalement, la partie communication du niveau infrastructure est utilisée directement par le niveau méta et indirectement par le niveau de base. En effet, le niveau de base peut accéder aux fonctionnalités de communication de l'infrastructure seulement à travers le gestionnaire de communication défini dans le niveau méta. Ce choix de conception permet de suspendre simplement les communications de base liées à un agent en cours d'adaptation. Le gestionnaire de communication est détaillé dans les paragraphes 5.1.3.4 et 5.2.

Sans perte de généralité, nous considérons que le niveau infrastructure d'un agent est implémenté comme un composant logiciel qui comprend trois interfaces : une interface de communication, une interface d'accès aux capteurs et une interface d'accès aux effecteurs.

5.1.2 Niveau de base

Le niveau de base définit le *comportement applicatif* de l'agent. Par applicatif, nous désignons le comportement qu'un agent doit manifester dans une application excepté le cas des opérations d'adaptations. Le niveau de base est composé d'un ensemble de composants utilisables par l'application. Selon la configuration utilisée, une partie de ces composants est impliquée dans l'assemblage de composants courants. En effet, l'assemblage de composants courant est la partie applicative réelle de l'agent ; d'autres composants contenus dans le niveau de base sont inutilisés mais peuvent être utilisés ultérieurement, c'est-à-dire après une adaptation.

Si cela est spécifié par la configuration courante, les composants de l'as-

semblage courant peuvent être connectés avec l'interface d'accès aux capteurs et l'interface d'accès aux effecteurs du niveau infrastructure. Concernant l'interface de communication, elle peut être connectée à l'assemblage courant uniquement à travers le gestionnaire de communication du niveau méta, comme déclaré dans le paragraphe 5.1.3.

5.1.3 Niveau méta

Le niveau méta est en charge du processus d'adaptation (déclenchement, décisions et exécution). Le cœur du niveau méta est le moteur d'assemblage MaDcAr (voir la sous-section 4.2.2). Les autres éléments de ce niveau sont : le gestionnaire de contexte de l'agent, la description du niveau de base, la politique d'assemblage, le gestionnaire de communication, le gestionnaire de contenu et la politique de gestion de contenu. Ces différents éléments sont représentés dans la figure 5.3.

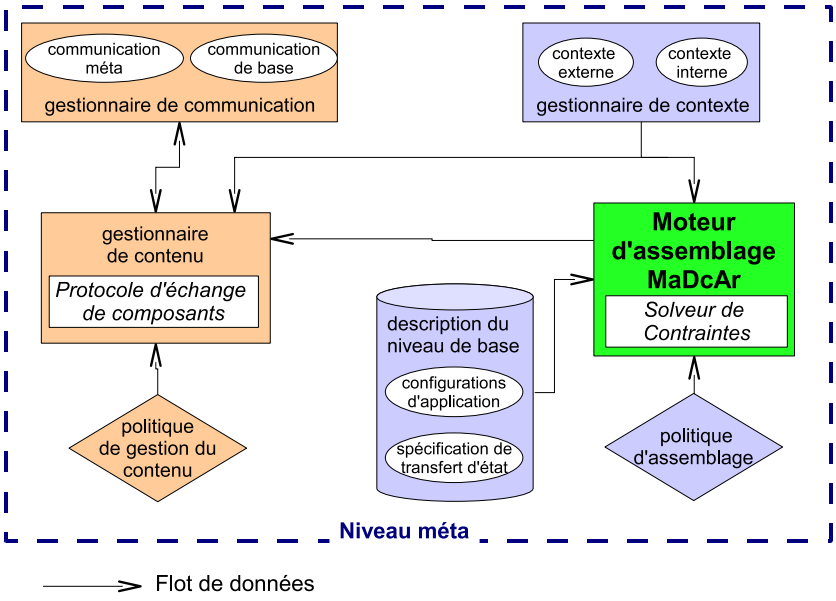

Figure 5.3 – Niveau méta de MaDcAr-Agent.

5.1.3.1 Le gestionaire de contexte de l'agent

Le gestionnaire de contexte de l'agent est similaire à ce qui est prévu pour MaDcAr (sous-section 4.2.3). Cependant, chaque agent est supposé avoir un capteur (`neighborhoodSensor`) dédié à la détection de l'apparition ou la disparition d'agents dans son voisinage. Le mécanisme sous-jacent

pour la détection de voisinage ainsi que la portée maximale de détection dépendent de l'infrastructure matérielle et logicielle sur laquelle l'agent est déployé. Rappelons que le contexte contient deux parties. Le *contexte interne* correspond aux informations connues sur le niveau de base (les composants, leurs attributs, etc.). Le *contexte externe* correspond aux informations perçues par l'intermédiaire du niveau infrastructure (par exemple, la localisation géographique de l'agent ou le voisinage de l'agent). Notons que contexte peut stocker des percepts ou des actions antérieures pour optimiser les actions futures. En particulier, le contexte externe de l'agent garde un historique des interactions les plus récentes avec les autres agents, par exemple pour ne pas leur demander plusieurs fois la même chose (voir le paragraphe 5.2.2.3).

5.1.3.2 La description du niveau de base de l'agent

La description du niveau de base représente des informations sur les « contenus » possibles du niveau de base, en termes de structures, de fonctionnalités et d'état. Cela correspond à ce que nous avons appelé une « description d'application » dans le modèle MADCAR (voir les sous-sections 4.2.4 et 4.3.1).

5.1.3.3 La politique d'assemblage de l'agent

La politique d'assemblage définit les données décisionnelles qui pilotent le processus de (ré)assemblage, en particulier la fonction de sélection SF_{bcc} qu'il faut optimiser pour choisir une configuration et des composants en fonction du contexte courant. Elle est décrite en détails dans la sous-section 4.2.5. Cependant, comme nous voulons permettre la conception d'agents ayant un très fort degré d'autonomie, nous considérons que chaque agent est une entité capable de s'auto-gérer et de prendre en compte le niveau des ressources dont il dispose pour décider de la fréquence de déclenchement des adaptations et de la durée des décisions.

Dans MADCAR-AGENT, (i) la période minimale de temps entre deux évaluations de la situation contextuelle (notée `minDurationBetweenAdaptationTriggering`) et (ii) la période maximale de temps allouée au processus de décision (notée `maxDurationForAdaptationDecision`) peuvent être spécifiées chacune par une fonction dont les paramètres sont les valeurs courantes des ressources matérielles, soit $value_{cpu}$, $value_{memory}$ et $value_{energy}$.

L'augmentation de la valeur de `maxDurationForAdaptationDecision` signifie que le processus d'assemblage a plus de temps pour décider de la meilleure adaptation à faire étant donné le contexte courant. Donc, l'aug-

mentation de la valeur de `maxDurationForAdaptationDecision` favorise l'optimalité du comportement applicatif de l'agent. De même, la diminution de la valeur de `minDurationBetweenAdaptationTriggering` signifie que l'agent a l'occasion de s'adapter plus souvent pour prendre en compte plus rapidement les changements de contexte. Donc, la diminution de la valeur de `minDurationBetweenAdaptationTriggering` favorise l'optimalité du comportement d'adaptation de l'agent. En revanche, l'augmentation de la valeur de `maxDurationForAdaptationDecision` et la diminution de la valeur de `minDurationBetweenAdaptationTriggering` impliquent une grande consommation de ressources matérielles par le processus d'adaptation. Par conséquent, le concepteur de l'agent doit définir ces deux fonctions avec précaution.

En dessous de certains niveaux de ressources matérielles, le processus de réassemblage est inutile voire dommageable. Par exemple, le manque de CPU durant le processus de décision, qui est limité dans le temps, peut mener à la formation d'un assemblage de composants très différent de l'assemblage optimal. Par conséquent, *il est recommandé d'effectuer des adaptations de manière moins fréquente lorsque les niveaux des ressources matérielles sont faibles.*

Cette « règle » doit être concrétisée par le concepteur d'agent lorsqu'il spécifie la politique d'assemblage, notamment les fonctions `minDurationBetweenAdaptationTriggering` et `maxDurationForAdaptationDecision`. Une telle règle empêche également un agent de s'adapter indéfiniment lors de situations critiques telles que le manque de CPU, de mémoire ou d'énergie. Donc, ces deux fonctions doivent prendre en compte ces différentes propriétés contextuelles.

5.1.3.4 Le gestionnaire de communication de l'agent

Le gestionnaire de communication de l'agent permet à un agent d'utiliser l'interface de communication du niveau infrastructure pour envoyer ou recevoir des messages. Le gestionnaire de communication intervient dans deux sortes de communications : les communications du niveau méta impliquant le gestionnaire de contenu et les communications du niveau de base impliquant l'assemblage courant.

Pour assurer que les deux niveaux de communication d'un agent n'interfèrent pas entre eux, le gestionnaire de communication associe la marque `meta` à chaque message provenant du moteur d'assemblage et la marque `base` sur chaque message provenant de l'assemblage de composants courant.

Enfin, le gestionnaire de communication contient un protocole d'interaction qui est utilisé par le niveau méta pour l'adaptation. Ce protocole d'inter-

actions pour l'adaptation, que nous nommons *interactions méta*, est partagé par tous les agents et nous le décrivons dans la sous-section 5.2.3. Pendant la réalisation d'une adaptation, les communications relatives au niveau de base (en particulier les communications entrantes) sont suspendues par le gestionnaire de communication. Le gestionnaire de communication stocke temporairement les messages bloqués et les réexpédie lorsque le processus d'assemblage se termine.

5.1.3.5 Le gestionnaire de contenu de l'agent

Le gestionnaire de contenu permet à un agent d'ajouter ou de supprimer des composants internes dans le niveau de base. Le gestionnaire de contenu peut accéder à certains résultats du moteur d'assemblage tels que les configurations prioritaires (voir la section 5.2.1), les composants les plus utilisés ou les composants les moins utilisés.

D'une part, il est possible de supprimer un composant interne inutilisé en cas de manque de mémoire. D'autre part, il est possible d'ajouter un composant interne s'il est susceptible d'améliorer les fonctionnalités de l'agent, notamment dans le cas de composants qui peuvent remplir des rôles qui ne peuvent être satisfaits par les composants déjà disponibles. Plus un agent possède de composants internes, plus il peut construire d'assemblages différents et plus il peut prendre en charge des situations contextuelles variées. Le gestionnaire de contenu est piloté par la politique de gestion de contenu.

5.1.3.6 La politique de gestion de contenu de l'agent

Le concepteur d'agent doit spécifier dans une *politique de gestion de contenu* quand et comment les composants internes doivent être ajoutés ou supprimés. Par exemple, il peut faire en sorte de supprimer en priorité les composants qui ont été les moins utilisés ou les composants qui requièrent la plus grande quantité de mémoire libre.

Une partie de la politique de gestion de contenu est dédiée aux préférences concernant les interactions méta. Le niveau méta d'un agent utilise un protocole d'interaction défini dans le gestionnaire de communication pour obtenir des composants qui sont requis pour les adaptations depuis d'autres agents. La problématique d'interactions méta est détaillé dans la sous-section 5.2.3.

5.2 Gestion de contenu dans MaDcAr-Agent

MaDcAr-Agent prend en charge la gestion du contenu en permettant l'ajout ou la suppression automatique de composants internes. Par ailleurs, ce

modèle prévoit que les agents utilisent des interactions au niveau méta pour améliorer leur adaptabilité. Il s'agit de permettre à chaque agent de déterminer dynamiquement des adaptations potentielles - mais impossible en l'état - et d'obtenir les composants nécessaires à ces adaptations en interrogeant d'autres agents. Ainsi, nous proposons de régir les échanges de composants entre agents d'une société d'agents par l'anticipation des adaptations que les agents sont susceptibles d'effectuer lorsqu'ils ont tous les composants requis.

Pour respecter au mieux l'autonomie de chacun des agents, l'échange de composants doit être le moins handicapant possible. Cela consiste à respecter certaines propriétés :

1. *équité* locale des échanges : il s'agit d'éviter qu'un agent passe son temps à envoyer des composants aux autres agents alors qu'il aurait besoin d'en demander ;
2. *non intrusivité* des échanges par rapport au comportement applicatif (*i.e.* de base) de l'agent : il s'agit d'éviter les échanges de composants lorsqu'un agent est trop occupé et de permettre aux agents de quitter simplement un échange en cours.

5.2.1 Politique de gestion du contenu

La politique de gestion du contenu de l'agent comporte un ensemble de données utilisées par le gestionnaire de contenu. Cet ensemble de données est employé par l'agent pour gérer son contenu, c'est-à-dire ajouter ou supprimer des composants de son niveau de base. L'ajout d'un composant nécessite que l'agent l'obtienne depuis d'autres agents.

La gestion de contenu est une préoccupation à la fois endogène et exogène. C'est pourquoi la politique de gestion du contenu est composée de deux parties. La gestion du contenu est traitée par chaque agent à la fois (1) localement par de simples ajouts et suppressions de composants et (2) à travers des interactions conduisant à l'échange de composants entre agents. Nous allons détailler ces deux composantes de la politique de gestion de contenu ci-dessous. La mise en œuvre de la politique de gestion du contenu est expliquée dans la sous-section 5.2.2.

5.2.1.1 Ajouts et suppressions de composants

La première partie de la politique de gestion du contenu est principalement utilisée par le gestionnaire de contenu pour *déterminer quand des composants peuvent être ajoutés ou supprimés, et quels composants doivent être ajoutés ou supprimés de manière prioritaire.* Elle est exprimée au travers des variables suivantes :

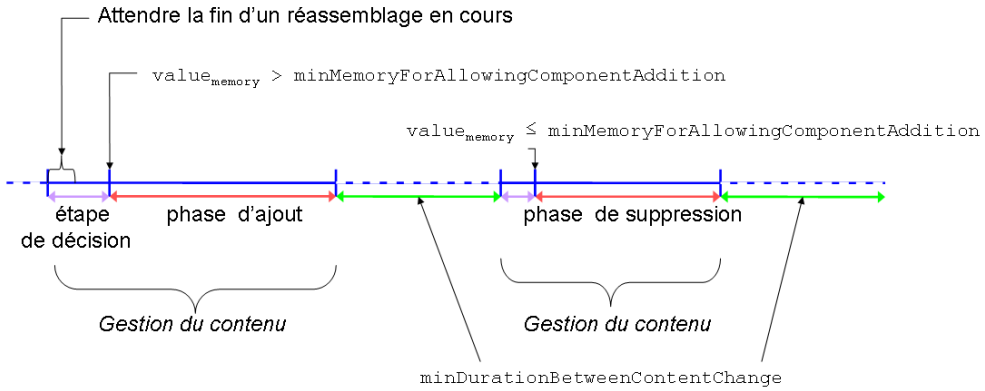

FIGURE 5.4 – Illustration des périodes de gestion du contenu dans MaDcAr-Agent.

- `minDurationBetweenContentChange` : la période minimale de temps entre deux sessions [1] de gestion de contenu ;
- `maxDurationForNeighborhoodUpdate` : la période maximale de temps pour mettre à jour le voisinage de l'agent de gestion de contenu ;
- `minMemoryForAllowingComponentAddition` : un seuil pour le niveau courant de mémoire libre sur l'infrastructure de déploiement au-dessus duquel l'ajout de composant est autorisé ;
- `componentUtilityFunction` : une fonction sur l'utilité d'un composant ;
- `minUtilityForDenyingComponentDeletion` : un seuil en-dessous duquel la valeur de `componentUtilityFunction` désigne un composant qui n'est plus assez utile, et pour lequel une suppression est donc autorisée.

La figure 5.4 montre un exemple de la manière dont ces variables influent sur l'enchainement des périodes de gestion du contenu d'un agent et les périodes sans gestion du contenu. On distingue deux sessions de gestion du contenu de l'agent. La première session induit une phase d'ajout de composants car la quantité de mémoire disponible est jugée suffisante ($value_{memory} >$ `minMemoryForAllowingComponentAddition`). La deuxième session induit une phase de suppression de composants, en supposant que les précédents ajouts de composants ont trop réduit la quantité de mémoire disponible ($value_{memory} \leq$ `minMemoryForAllowingComponentAddition`). Après chaque session de gestion du contenu, l'agent a du temps qu'il peut consacrer entièrement à l'exécution du niveau de base (*i.e.* la partie applicative) et aux

1. Une session délimite le temps pendant lequel le gestionnaire de contenu décide puis réalise des ajouts/suppressions de certains composants.

réassemblages. La section 5.2.2 explique de manière détaillée ce processus de gestion du contenu.

La valeur de `componentUtilityFunction` dépend à la fois de l'« utilité passée » (basée sur des taux d'utilisation) et l'« utilité future » (basée sur des évaluations de l'intérêt) des composants présents dans le niveau de base de l'agent. Pour cela, le gestionnaire de contenu peut accéder au contexte de l'agent et à certains résultats du moteur d'assemblage (meilleures configurations pour chaque situation contextuelle rencontrée par l'agent, liste des composants les plus utilisés, liste des composants les moins utilisés,...). Un exemple de fonction d'utilité est donné par la formule 5.1. Le taux d'utilisation passé d'un composant (*pastUseRate*) est le ratio entre son temps d'utilisation cumulée dans un assemblage et son temps de présence cumulée dans l'agent. L'utilité future d'un composant (*futureUsefulness*) est une fonction qui détermine si oui ou non un composant va améliorer les fonctionnalités de l'agent. Pour ce faire, il suffit de tester à quel point il satisfait les rôles des « configurations prioritaires », c'est-à-dire les configurations qui étaient les plus pertinentes pour certaines situations contextuelles passées mais qui ne sont pas éligibles. Par exemple, nous pouvons définir l'utilité future d'un composant de la manière suivante : 100 s'il satisfait un rôle non minimalement satisfait [2] dans une configuration prioritaire, 50 s'il satisfait un rôle qui est au moins minimalement satisfait dans une configuration prioritaire, et 0 sinon.

$$\begin{aligned} componentUtilityFunction(component, context) &= 60 * pastUseRate(component, context) \\ &\quad +40 * futureUsefulness(component, context) \end{aligned} \tag{5.1}$$

5.2.1.2 Échange de composants entre agents

La deuxième partie de la politique de gestion du contenu est utilisée par le gestionnaire de communication pour *déterminer si un agent est disposé à échanger des composants, et quels sont ces composants.* Cela consiste en la spécification de :

- `maxDurationForWaitingAgentResponse` : une limite de temps d'attente pour la communication entre agents durant un échange de composants ;
- `agentSociabilityFunction` : une fonction booléenne pour la capacité sociale de l'agent, c'est-à-dire sa tendance à traiter ou à ignorer des demandes de composants entrantes de la part d'autres agents ;

2. Un rôle $R(min, max)$ est dit minimalement satisfait lorsqu'il y a au moins min composants disponibles qui sont compatibles avec R.

– `rolePriorityFunction` : une fonction permettant de déterminer pour quels rôles la recherche de composants est prioritaire ;
– `maxSizeOfPrioritaryRolesSet` : une taille maximale pour l'ensemble contenant les rôles prioritaires, c'est-à-dire pour lesquels il faut demander des composants.

Un exemple pour `rolePriorityFunction` est la fonction qui retourne 100 dans le cas d'un rôle non minimalement satisfait appartenant à une configuration prioritaire, 50 dans le cas d'un rôle au moins minimalement satisfait appartenant à une configuration prioritaire, et 0 dans les autres cas. La valeur de `agentSociabilityFunction` doit être à « faux » lorsque l'agent manque de ressources matérielles telles que la CPU. De plus, pour empêcher des échanges de composants simultanés, la valeur de `agentSociabilityFunction` est mis à « faux » lorsque l'agent est déjà en train d'entreprendre un échange de composants avec un autre agent. Cela empêche que les échanges de composants ne monopolisent trop de ressources chez un agent.

5.2.2 Processus de gestion du contenu

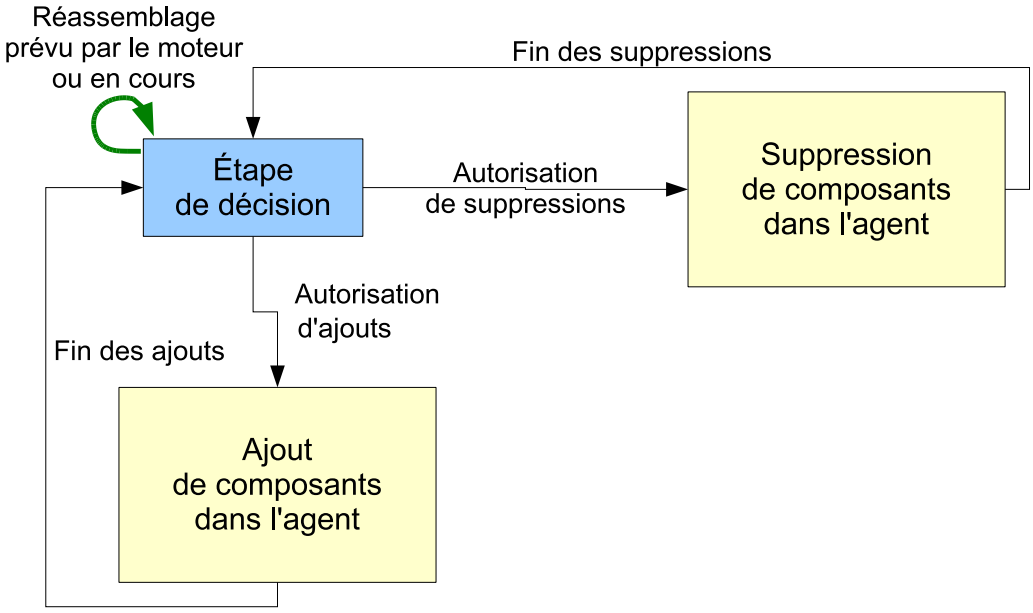

Figure 5.5 – Processus de gestion du contenu (vue simplifiée).

Le processus de gestion du contenu d'un agent permet d'ajouter, de supprimer ou d'échanger des composants selon la politique de gestion du contenu définie par le concepteur de l'agent (voir la sous-section 5.2.1).

Il se compose de plusieurs phases : l'étape de décision, la suppression de composants et l'ajout de composants. La première phase est le point d'entrée du processus et intervient à la fin de chaque période de temps (`minDurationBetweenContentChange`) prévue entre deux « sessions » de gestion de contenu, que ce soit une session dédiée à l'ajout de composants ou bien à la suppression de composants.

5.2.2.1 Étape de décision

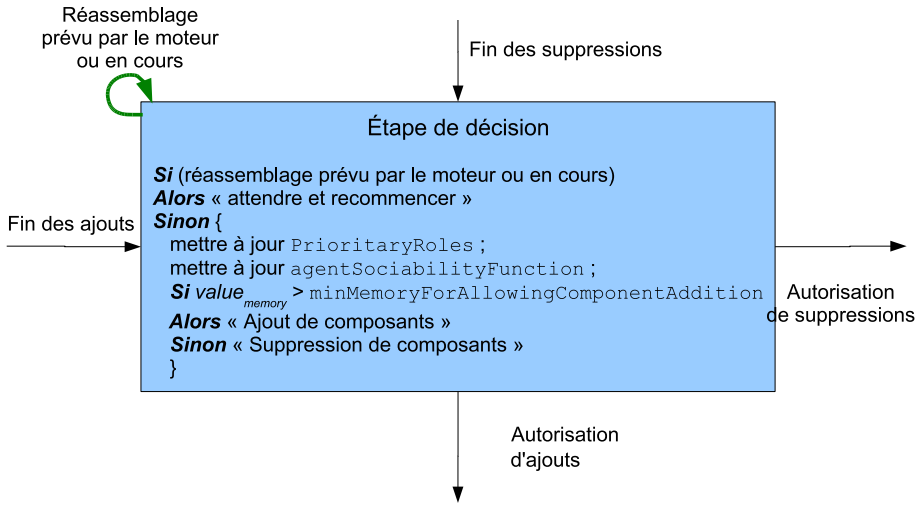

FIGURE 5.6 – Étape de décision de la gestion du contenu.

L'étape de décision permet principalement de *décider s'il faut ajouter ou supprimer des composants* au cours de la session de gestion du contenu. Le gestionnaire du contenu vérifie s'il peut ajouter ou supprimer des composants en testant la quantité de mémoire disponible sur l'infrastructure de déploiement. Si $value_{memory} \leq$ `minMemoryForAllowingComponentAddition`, alors certains composants doivent être supprimés et il est interdit d'ajouter de nouveaux composants dans l'agent. Si $value_{memory} >$ `minMemoryForAllowingComponentAddition`, alors des composants peuvent être ajoutés.

Le déroulement de l'étape de décision est présenté par la figure 5.6. La première chose à faire est de tester l'état du moteur d'assemblage pour que le processus de gestion du contenu et le processus de réassemblage ne se perturbent pas mutuellement. Pour éviter les incohérences, les deux processus doivent être coordonnés. Nous donnons la priorité au processus de réassemblage pendant l'étape de décision. C'est pourquoi, l'étape de décision est

« bloquée » lorsqu'un réassemblage de composants est prévu (i.e. décidé) ou en cours de réalisation. Dans le cas contraire, c'est le processus de gestion du contenu qui devient prioritaire, de manière à empêcher la réalisation des réassemblages pendant la suppression de composants.

Avant de décider d'ajouter ou de supprimer des composants dans l'agent, il faut mettre à jour certaines valeurs : `PrioritaryRoles` et `agentSociabilityFunction`. `PrioritaryRoles` est une collection ordonnée contenant des rôles issus des configurations prioritaires. La taille maximale de `PrioritaryRoles` est `maxSizeOfPrioritaryRolesSet`. Un agent utilise cette collection pour demander des composants à d'autres agents. Le critère d'ordre de `PrioritaryRoles` est donné par le concepteur de l'agent à travers la fonction `rolePriorityFunction`.

5.2.2.2 Suppression de composants

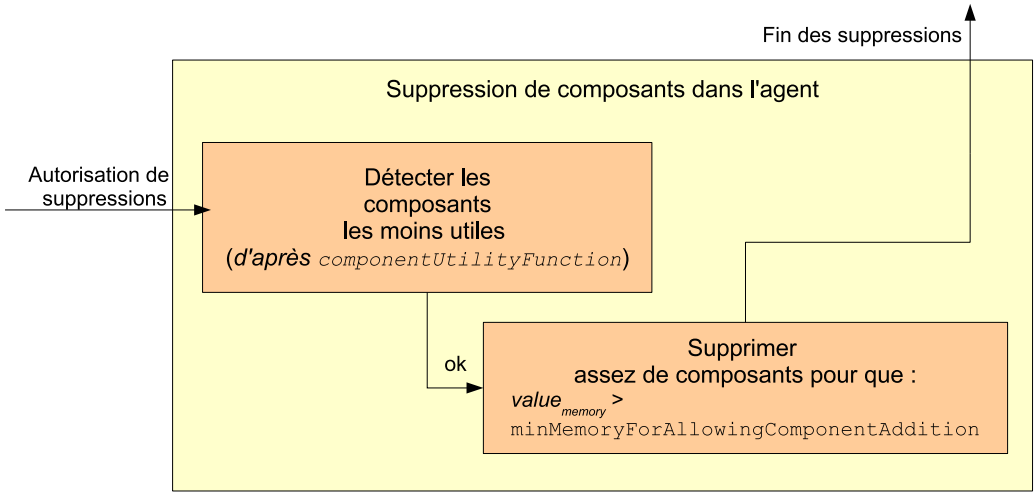

Figure 5.7 – Partie « suppression de composants » de l'automate.

Cette étape permet essentiellement de *supprimer des composants* de l'agent. Le déroulement de la suppression de composants est présenté par la figure 5.7. Le gestionnaire de contenu détecte les composants les moins utiles pour l'agent d'après `componentUtilityFunction` et `minUtilityForDenyingComponentDeletion`. Ensuite, les composants les moins utiles sont supprimés[3] successivement jusqu'à

3. Il n'y a pas de vérification que les composants à supprimer ne serait pas utiles à d'autres agents, mais il est clair qu'une telle vérification par un agent serait profitable pour le système global mais au détriment de l'urgence de réaction pour cet agent.

ce que le niveau de mémoire libre devienne supérieur au seuil `minMemoryForAllowingComponentAddition`. Notons que les composants sont nécessairement inutilisés (*i.e.* pas dans l'assemblage courant) au moment où ils sont supprimés.

5.2.2.3 Ajout de composants

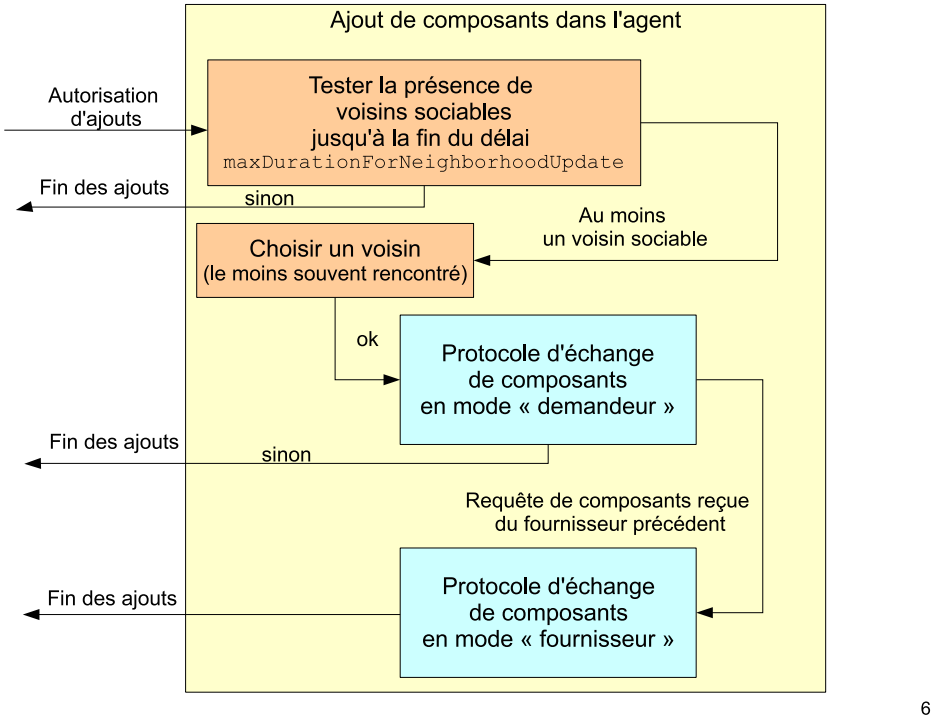

FIGURE 5.8 – Partie « ajout de composants » de l'automate.

Cette partie permet essentiellement d'*ajouter des composants* dans l'agent. Le déroulement de l'ajout de composants est présenté par la figure 5.8.

L'agent doit d'abord tester la présence de voisins acceptant de traiter les demandes de composants, c'est-à-dire les voisins pour lesquels la valeur de `agentSociabilityFunction` est à « vrai ». Cette information est obtenue avec la sonde `neighborhoodSensor`. L'exploration du voisinage est limitée par le délai `maxDurationForNeighborhoodUpdate`. Si aucun voisin valable n'a été identifié passé ce délai, alors la phase d'ajout de composants se termine immédiatement. Sinon, il faut choisir un des voisins, de préférence un agent

qui n'a pas ou peu interagi avec l'agent précédemment. Cela est possible car le contexte externe de l'agent garde un historique des échanges les plus récents (voir le paragraphe 5.1.3.1).

Chaque fois qu'un agent voisin est choisi, l'agent doit démarrer une interaction pour demander des composants qui satisfont (au moins) un rôle parmi `PrioritaryRoles` et attend que ce voisin réponde jusqu'à un certain délai de dépassement (`maxDurationForWaitingAgentResponse`). Si un agent voisin envoie un composant, alors l'agent receveur doit l'accepter. Par ailleurs, si l'agent reçoit à son tour une demande de composants de la part de l'agent qui vient de lui fournir des composants, alors il est tenu de traiter sa requête. Dans la section suivante, nous allons détailler les interactions méta au service de l'adaptation.

5.2.3 Interactions méta au service de l'adaptation

Dans MaDcAr-Agent, des interactions méta entre agents offrent la possibilité aux agents d'obtenir des composants pour pouvoir construire de nouveaux assemblages.

5.2.3.1 Principe

Dans le modèle MaDcAr-Agent, l'organisation sous-jacente pour les interactions méta[4] est du style *pair-à-pair* car elle est facile à obtenir et elle satisfait les propriétés d'ouverture et d'autonomie parmi les agents. En effet, dans notre approche, un agent ne connaît pas *a priori* les autres agents, ni leurs fonctionnalités. Lorsqu'un agent a besoin d'obtenir une fonctionnalité externe, *a fortiori* un composant, il interroge les agents de son voisinage jusqu'à recevoir le composant requis. Ce fort découplage entre les agents vise à faciliter l'ajout, la suppression ou l'adaptation d'un agent sans déranger les autres agents du système.

Les interactions méta sont destinées à optimiser l'autonomie des agents concernant l'adaptation. Si les agents ont peu de composants à leur disposition alors leur capacité à s'adapter à des changements environnants ne pourra qu'être limitée. Par conséquent, les agents doivent être capables d'acquérir de nouveaux composants de manière autonome, *i.e.* sans intervention humaine, grâce à des *capacités d'interaction* : (1) détecter l'apparition et la disparition d'agents dans un voisinage avec une sonde, (2) interroger un agent sur les composants qu'il possède et (3) échanger des données (composants ou rôles) avec un agent voisin.

4. La définition d'une organisation pour des raisons applicatives (donc, concernant le niveau de base des agents) va au-delà des objectifs de ce modèle d'agent.

5.2.3.2 Protocole d'échange de composants

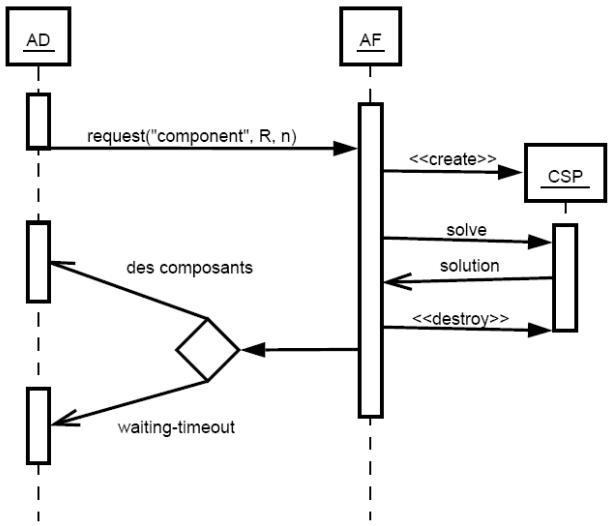

FIGURE 5.9 – Protocole d'échange de composants entre deux agents.

Le cœur de la stratégie d'interaction est le *protocole d'échange de composants*, illustré par la figure 5.9. Il est défini dans le gestionnaire du contenu inclus dans le niveau méta des agents. Chaque agent peut être vu comme un fournisseur de composants, ainsi qu'un demandeur de composants. Le processus d'interaction entre un agent demandeur AD et un agent fournisseur AF est le suivant :

- **Sélection d'un fournisseur :** AD sélectionne un agent AF parmi ses voisins (le moins souvent rencontré).
- **Protocole d'échange :**
  1. AD émet une requête de demande de (au plus) n composants compatibles avec le rôle R appartenant à `PrioritaryRoles`.
  2. Si la valeur courante de `agentSociabilityFunction` est à « faux » pour AF alors il rejette la demande, ce qui met fin au protocole d'échange. Par contre, si la valeur courante de `agentSociabilityFunction` est à « vrai » quand AF reçoit cette requête[5] construit un CSP (*i.e.* un problème de résolution de contraintes) à partir des spécifications du rôle R et cherche des

5. Notons que si un agent reçoit des requêtes issues de plusieurs agents, alors il choisit de préférence de traiter la demande d'un agent avec qui il a peu (ou pas) interagit précédemment.

composants qui peuvent remplir le rôle R parmi tous ses composants inutilisés.

3. AF répond à AD en envoyant successivement des composants (au plus n) qui satisfont le rôle R et pour lesquels la valeur de `componentUtilityFunction` est inférieure à `minUtilityForDenyingComponentDeletion`. Pendant cette étape (et jusqu'à la fin du protocole d'échange), la valeur de `agentSociabilityFunction` est considéré à « faux » pour ignorer les éventuelles requêtes concurrentes.
4. L'échange de composants se termine à la fin du délai `maxDurationForWaitingAgentResponse`, en comptant à partir de la dernière réception d'un composant. En d'autres termes, un échange de composant est fini lorsque AF a envoyé tous les composants qu'il voulait envoyer ou bien lorsque deux agents ne sont plus voisins (par exemple, à la suite d'une déconnexion). Cependant, pour avoir l'*équité* dans le système multi-agent, ce protocole doit garantir que chaque agent a l'opportunité d'alterner entre la « tâche de fournisseur de composants » et la « tâche de demandeur de composants ». Pour prendre en compte cette priorité, il suffit de mémoriser l'identifiant des agents pour au moins les deux échanges précédents. Dans ce cas, si AD a reçu au moins un composant de AF, alors l'éventuelle requête de demande de AF vers AD est prioritaire sur n'importe quelle demande envoyé à AD.

– **Intégration des composants :** AD place les composants reçus dans l'ensemble des composants inutilisés dans le niveau de base.

Les nouveaux composants obtenus à la fin d'un échange entre agents sont susceptibles d'être utilisés lors d'un prochain réassemblage.

5.3 Formulation d'architectures d'agents avec MaDcAr-Agent

Dans cette section, nous montrons que MaDcAr-Agent permet de concevoir des comportements d'agents selon différentes architectures.

5.3.1 Cas d'école : exploration d'un labyrinthe

Pour illustrer notre approche, nous allons nous appuyer sur un exemple simplifié : l'exploration d'un labyrinthe par un agent. Le labyrinthe est formé

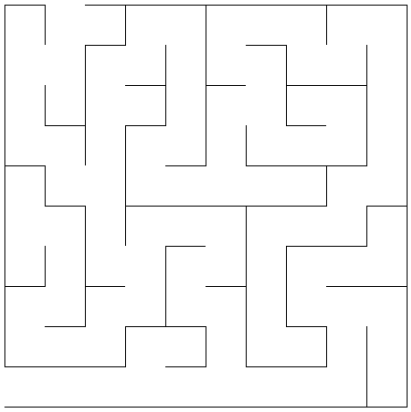

FIGURE 5.10 – Exemple de labyrinthe à explorer.

de cellules carrées séparées ou non par des murs, comme le montre la figure 5.10. Considérons plusieurs configurations qui utilisent des architectures différentes pour exécuter un comportement d'exploration de labyrinthe. La première configuration effectue l'exploration selon une architecture réactive. La deuxième configuration effectue l'exploration selon une architecture délibérative (BDI).

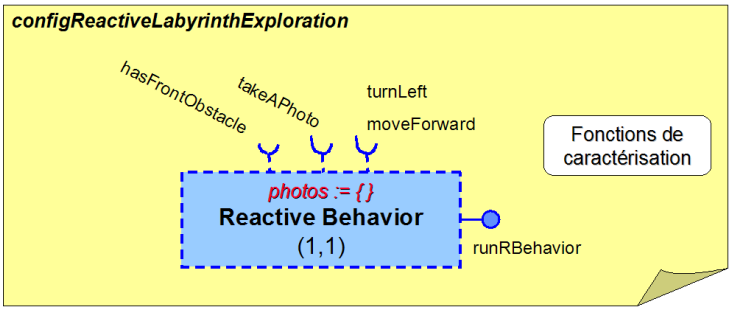

FIGURE 5.11 – Configuration `configReactiveLabyrinthExploration`.

Le comportement associé à la première configuration est donné par la figure 5.13. Le comportement associé à la deuxième configuration est donné par la figure 5.14. Ces configurations sont représentées respectivement par les figures 5.11 et 5.12. À l'exception de `Plans`, chaque rôle impliqué requiert un seul composant, car les multiplicités minimales et maximales sont à 1. En particulier, `Beliefs` et `Desires` décrivent chacun un composant ayant un attribut de type collection, contenant les croyances générées par le comportement délibératif et les désirs de l'agent. En revanche, d'après les multiplicités du rôle `Plans`, il y a un composant par plan. Chaque plan expose le

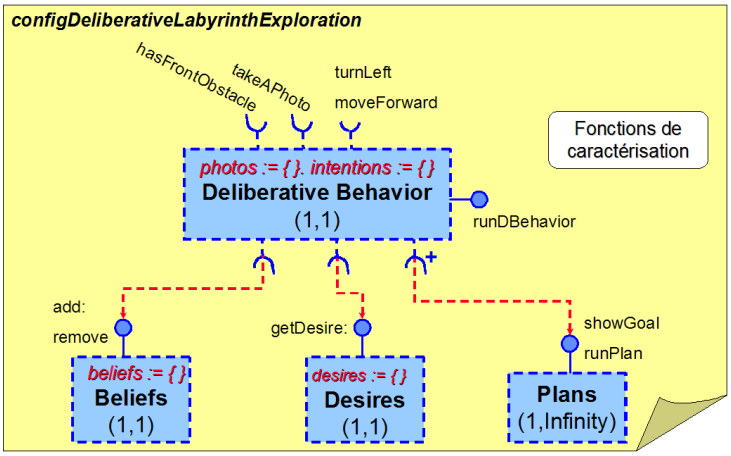

Figure 5.12 – Configuration `configDeliberativeLabyrinthExploration`.

but qu'il se fixe. De cette manière, le comportement délibératif peut trouver un plan en fonction du désir qu'il veut poursuivre. Il est sous-entendu que les interfaces requises présentes dans les configurations sont connectées au composant du niveau infrastructure qui doit fournir notamment les fonctionnalités de détection d'obstacles et de déplacement de l'agent.

```
"---------- ReactiveBehavior ----------"
SI il y a un obstacle frontal ALORS {
  Prendre une photo.
  Tourner à gauche }
SINON
  Avancer tout droit d'un pas.
```

Figure 5.13 – Pseudo-code du comportement réactif.

5.3.2 Adaptation et transfert d'état

MaDcAr-Agent est un modèle d'agent qui permet au concepteur de se détacher d'une architecture d'agent particulière. Celui-ci a la possibilité de définir plusieurs configurations alternatives pour un même comportement applicatif.

Pour simplification, les figures 5.11 et 5.12 ne détaillent pas les fonctions de caractérisation utilisées. Nous les avons définies de telle sorte que les deux configurations soient pertinentes pour les mêmes situations contextuelles. Toutefois, il est clair que la configuration symbolisant une architecture délibérative est plus gourmande en ressources matérielles que celle symbolisant une architecture réactive, car le processus de décision de cette dernière est

```
"---------- DeliberativeBehavior ----------"
Générer des croyances à partir des différents capteurs.
SI il n'y a pas de plan courant ALORS {
  Choisir un désir à poursuivre.
  Trouver un plan pour atteindre ce désir }
Décider de la prochaine action à faire (intention) selon le plan.
Réaliser cette action.
Tout répéter en vérifiant que le plan est toujours valide.
"---------- Desires ----------"
{ Explorer le labyrinthe.
  Prendre des photos à chaque obstacle }
"---------- Plans ----------"
POUR Explorer le terrain.
IL FAUT Avancer droit jusqu'à atteindre un obstacle.
POUR Traiter un obstacle
IL FAUT Prendre une photos, puis Changer de direction.
. . .
```

Figure 5.14 – Pseudo-code du comportement délibératif.

beaucoup moins complexe. Ainsi, MaDcAr-Agent permet de passer automatiquement d'une architecture à l'autre en réalisant des réassemblages guidés par une politique d'assemblage (non représentée) qui veut par exemple que la deuxième configuration ne soit utilisée que lorsque les ressources matérielles disponibles sont abondantes.

Par ailleurs, les changements automatiques de configurations requièrent de transférer automatiquement la valeur de certains attributs entre deux configurations. La définition d'un réseau de transfert d'état (voir la sous-section 4.3.3) évite les pertes d'informations lors des changements de configurations.

Celui de la figure 5.15 comporte six nœuds représentant les quatre attributs variables (ceux dont les valeurs sont modifiées par les composants) et deux attributs virtuels (ceux qui n'appartiennent pas à une configuration). Notons que l'attribut `desires` est fixe, ce qui signifie que l'ensemble des désirs de l'agent est considéré comme immuable dans cet exemple. C'est pourquoi il ne figure pas dans le réseau de transfert d'état (voir la section 4.3). Chacun des nœuds a comme fonction d'initialisation la création d'une collection ordonnée. Les variables `photos` sont équivalentes dans les deux configurations : il s'agit de l'ensemble des photos prises en rencontrant des obstacles, d'où une fonction identité dans les deux sens du transfert. L'attribut présent dans la configuration `configDeliberativeExploration` est défini comme maître de l'autre (choix arbitraire). En revanche, les variables `beliefs` et `intentions` ne sont présentes que dans la configuration `configDeliberativeExploration`. Leurs valeurs doivent être non persis-

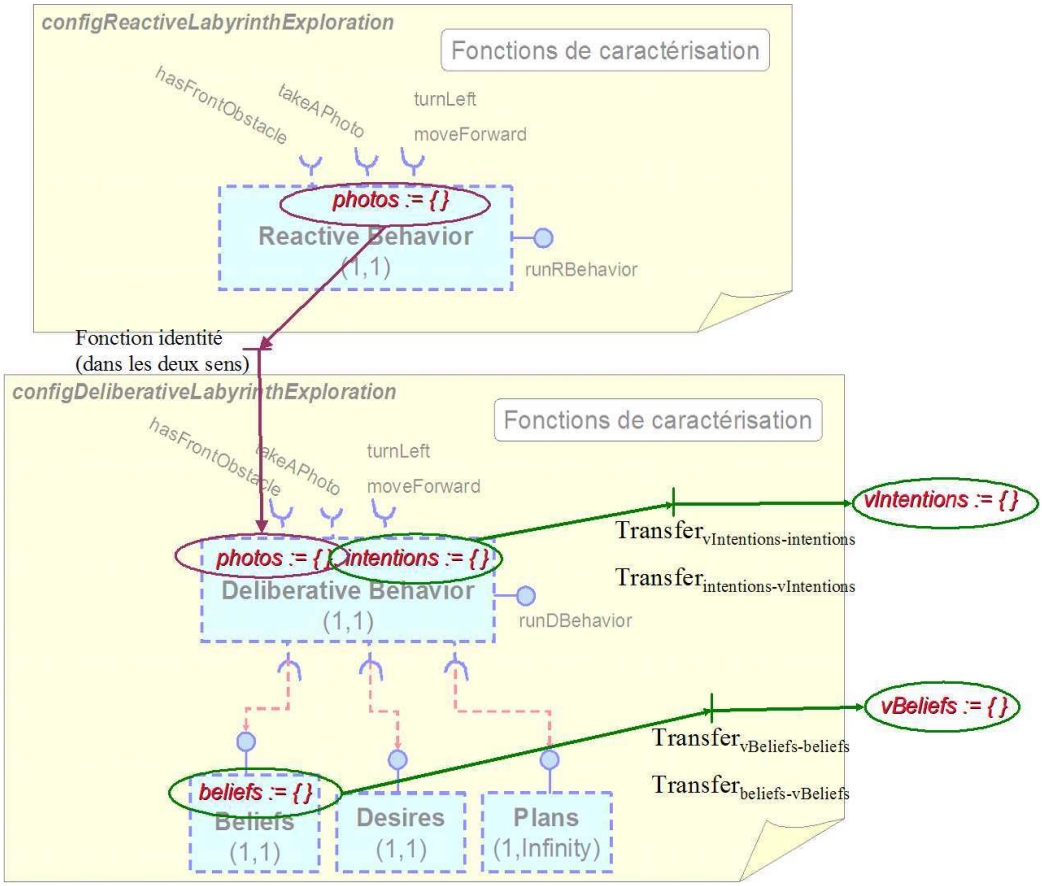

FIGURE 5.15 – Réseau de transfert d'état pour l'exploration d'un labyrinthe.

tantes lorsque la configuration change car les croyances et les intentions qu'elles contiennent risquent de devenir obsolètes. C'est pourquoi il y a deux attributs virtuels maîtres qui stockent leurs valeurs et les restituent si et seulement si la nouvelle configuration est égale à l'ancienne configuration.

Par exemple, les fonctions de transfert pour les intentions sont définies par le pseudo-code de la figure 5.16.

```
"---------- Transfer vIntentions-intentions ----------"
cInt = gestionnaireDeContexte.contexteInterne().
intentions.reinitialiser().
SI (cInt.configurationAncienne() = cInt.configurationNouvelle()) ALORS
  intentions.ajouterTous( vIntentions.valeurs() )
SINON vIntentions.reinitialiser().
"---------- Transfer intentions-vIntentions ----------"
vIntentions.reinitialiser().
vIntentions.ajouterTous( intentions.valeurs() ).
```

Figure 5.16 – Pseudo-code du transfert d'état pour les intentions.

5.4 Évaluation et comparaison avec l'état de l'art

Notre démarche est d'analyser notre modèle avec la même grille d'évaluation que celle utilisée pour l'état de l'art (voir le chapitre 3).

5.4.1 Degré d'adaptabilité de MaDcAr-Agent

Présentation MaDcAr-Agent est un modèle d'agents auto-adaptables à base de composants. Les mécanismes d'adaptation mis en œuvre reposent sur le modèle de moteurs d'assemblage MaDcAr et les décisions d'adaptation sont basées sur un solveur de contraintes.

Conception d'un agent Avec MaDcAr-Agent, un concepteur d'agent doit insérer des configurations et des composants pour délimiter les assemblages autorisés (structure et comportement). D'une part, les configurations représentent des comportements complexes réutilisables selon la situation contextuelle courante et les composants disponibles. D'autre part, les configurations attribuent une architecture particulière (réactive, délibérative, ad hoc, ...) à chacun de ces comportements. En outre, ces agents peuvent gérer leur contenu en supprimant des composants ou en s'échangeant des composants selon leurs besoins. C'est pourquoi, le concepteur doit définir des

politiques pour contrôler quand et comment doivent s'opérer les mécanismes d'adaptation et de gestion du contenu.

Adaptation d'un agent Les adaptations peuvent être déclenchées par l'agent de manière régulière, lorsqu'il existe un meilleur assemblage de composants internes que l'assemblage courant. L'utilisation d'un solveur de contraintes dans le processus de décision mis en œuvre dans le moteur d'assemblage permet d'éliminer rapidement les configurations qui ne sont pas éligibles ou qui sont peu pertinentes, et de sélectionner une configuration et des composants qui sont les meilleurs d'après la fonction de sélection issue de la politique d'assemblage et du temps réservé pour les opérations d'adaptation. De plus, l'agent essaie en parallèle de gérer son stock de composants en faisant des échanges avec d'autres agents pour d'une part, (1) obtenir et garder les composants les plus utiles pour lui et d'autre part, (2) transmettre aux autres agents des composants qu'il n'utilise pas ou les supprimer si nécessaire. Nous présentons ci-dessous la caractérisation de l'adaptation d'un agent selon les critères définis dans la sous-section 3.1.3, d'où le tableau 5.1.

- *Portée :* la cible de l'adaptation est un assemblage de composants : donc elle porte à la fois sur le comportement et l'architecture de l'agent.
- *Initiateur :* l'agent (par son moteur d'assemblage) peut déclencher les adaptations.
- *Moment :* le déclenchement des adaptations est fonction de la politique d'assemblage et du contexte d'exécution qui est obtenu d'après le gestionnaire de contexte.
- *Mise en œuvre :* l'architecture de l'agent (niveau de base) est dynamiquement adaptable grâce au moteur d'assemblage (niveau méta) qui révise les composants et leurs connexions dans l'agent.
- *Reconfigurabilité :* l'infrastructure d'adaptation (*i.e.* le niveau méta d'un agent) est facilement reconfigurable en modifiant la politique d'assemblage et la politique de gestion du contenu de l'agent. De plus, les politiques de l'agent prennent en compte l'état des ressources de l'agent pour influencer automatiquement la manière de réaliser les adaptations (notamment la durée des sessions de réassemblage et de gestion du contenu).

Portée	**Initiateur**	**Moment**	**Mise en œuvre**	**Reconfigurabilité**
+++	+++	+++	+++	+++

TABLE 5.1 – Critères de l'adaptation pour MADCAR-AGENT

Évaluation Notre approche se caractérise par l'utilisation d'un niveau méta qui est configurable à travers deux politiques d'adaptation : l'une centrée sur le réassemblage des composants et l'autre centrée sur les ajouts et suppressions de composants. De plus, l'utilisation d'un solveur de contraintes pour choisir une des configurations prévues par le concepteur en fonction des composants disponibles et de la situation contextuelle courante assure que les opérations d'adaptation sont correctes. Un agent n'est pas complètement réflexif dans ce modèle dans la mesure où il n'est pas possible d'adapter automatiquement le niveau méta (gestionnaire de contexte, moteur d'assemblage, etc.). Néanmoins, l'agent dispose des mécanismes nécessaires pour modifier le comportement et la structure de son niveau de base. Enfin, les principaux éléments définissant un agent (configurations et politiques) sont assez abstraits pour être réutilisée dans un agent doté de composants différents du précédent.

5.4.2 Comparaison de MaDcAr-Agent avec l'état de l'art

Dans la plupart des modèles d'agents, l'adaptation des agents n'est pas traitée spécifiquement. Il est vrai que les agents se comportent généralement en fonction du contexte qu'ils perçoivent. Mais, lorsque les changements de contexte apparaissent de manière non anticipée, on n'a aucune assurance que l'agent puisse s'accommoder de toutes les situations sans changer sensiblement sa structure et son comportement. Certaines architectures d'agents à couches, notamment l'architecture Meta-Control ainsi que les architecture hybrides TouringMachine et InterRRap, ont proposé des mécanismes dédiés à l'auto-adaptation. Néanmoins, les adaptations prises en charge ne concernent que le comportement de l'agent. La rigidité de la structure des agents restreint leur domaine d'utilisation et rend difficile leur réutilisation dans un cadre sensiblement différent, d'autant plus que ces architectures ne reposent pas sur des composants logiciels. Récemment, plusieurs modèles d'agents auto-adaptables se sont appuyés sur une architecture plus flexible, en particulier des modèles d'agents à base de composants : MAST, MALEVA, Magique, BOND et JavAct$^{\delta}$. Le tableau 5.2 récapitule les évaluations de MaDcAr-Agent et d'autres modèles d'agents auto-adaptables.

MaDcAr-Agent va plus loin que la plupart des travaux existants en définissant un modèle pour des agents capables de déclencher et réaliser automatiquement des adaptations en fonction de leurs besoins, tout en maintenant un contrôle sur les modifications à effectuer[6]. Dans MAST, le dé-

6. En spécifiant grâce aux configurations les changements qui sont autorisés ou pas.

	Portée	Initiateur	Moment	Mise en œuvre	Reconfigurabilité
Modèles d'agents sans composants					
TouringMachine	+	+++	+++	+	+
InteRRap	+	+++	+++	+	+
Meta-Control	++	+++	+++	++	++
Modèles d'agents à base de composants					
MAST	+++	+	+++	++	+
Magique	+++	+++	+++	++	+
MALEVA	+++	+++	+++	++	+
BOND	+++	+++	+++	++	++
JavAct$^{\delta}$	+++	+++	+++	+++	++
Notre proposition					
MaDcAr-Agent	+++	+++	+++	+++	+++

Table 5.2 – Récapitulatif des modèles d'agents auto-adaptables

clenchement d'une adaptation nécessite une intervention humaine (ajout ou suppression d'un composant). Dans MALEVA, les adaptations sont effectuées par un composant spécifique, la tâche du concepteur de l'agent se résument à associer un assemblage prédéfini à certains évènements pour le déclenchement des adaptations. Dans Magique, les agents s'adaptent plus dans une perspective de répartition de charge au niveau d'un système multi-agent (par échange de composants) que pour les besoins propres des agents. Dans BOND, les adaptations sont programmées par des automates qui semblent complètement spécifiques à l'application courante et les critères d'adaptations manquent d'abstraction. Dans MAST, MALEVA et BOND, les propriétés extra-fonctionnelles comme les prises en compte des ressources matérielles ne prennent pas place dans le processus d'adaptation.

Le degré de configurabilité de MaDcAr-Agent et de JavAct$^{\delta}$ est supérieur aux autres. JavAct$^{\delta}$ met l'accent sur le maintien stricte de la cohérence des adaptations, basée sur un outil qui vérifie statiquement les spécifications du concepteur pour s'assurer que les adaptations potentielles vont terminer correctement. Par contre, MaDcAr-Agent vise une gestion plus souple de la cohérence des adaptations, basée sur l'utilisation d'un solveur de contraintes qui assure dynamiquement que les adaptations effectuées satisfont les contraintes requises, c'est-à-dire les spécifications du concepteur. De plus, les capacités d'adaptation des agents sont définies au niveau modèle à travers des concepts de haut niveau (configuration, rôle et politique) qui favorisent notamment la séparation des préoccupations, la réutilisabilité et la maintenance des agents.

Pour finir, MaDcAr-Agent est le seul modèle d'agent (à notre connaissance) qui vise à permettre à un concepteur d'agent aussi bien l'utilisation d'architectures classiques d'agents (réactifs, délibératifs, ...) que la définition d'architectures plus spécifiques à l'application visée.

5.5 Conclusion

Dans ce chapitre, nous avons présenté le modèle MaDcAr-Agent destiné à la conception d'agents auto-adaptables dans un SMA ouvert. Ce modèle permet au concepteur d'un agent de spécifier à travers des concepts de haut-niveau (rôle, configuration, politique) quand et comment l'agent doit s'adapter. Les adaptations d'agent peuvent aussi bien concerner le comportement de l'agent que son architecture.

MaDcAr-Agent se compose de trois niveaux : un niveau infrastructure servant d'interface avec le monde extérieur (communication, capteurs, effecteurs), un niveau de base qui correspond à la partie applicative de l'agent et un niveau méta qui gère le niveau de base. Le niveau méta intègre notamment un moteur d'assemblage automatique et dynamique de l'agent, ainsi qu'une description du niveau de base de l'agent. La description correspond à des configurations (*i.e.* spécification d'un ensemble d'assemblages valides) et des relations pour le transfert d'état entre ces configurations. Par ailleurs, le niveau méta permet de spécifier des politiques pour traiter différents aspects de l'agent. Une politique d'assemblage dirige le moteur d'assemblage pour notamment déclencher et réaliser les adaptations, en choisissant une configuration et des composants en fonction de la situation contextuelle courante (ressources matérielles disponibles, agents voisins, etc.). Une politique de gestion du contenu dirige le gestionnaire de contenu de l'agent pour, en fonction de ses besoins, supprimer des composants ou en obtenir de nouveaux depuis d'autres agents.

La distinction entre la description applicative de l'agent (description du niveau de base) et la politique d'assemblage permet une séparation explicite entre le « comportement normal » de l'agent et son « comportement d'adaptation ». Cette modularité simplifie à la fois la conception et l'évolution des agents.

La possibilité d'échanger des composants entre agents permet d'améliorer leur degré d'autonomie en ce qui concerne les adaptations. En effet, puisque les agents peuvent obtenir des composants sans intervention humaine, alors ils sont capables de s'adapter à une plus grande variété de contextes. Par conséquent, un concepteur d'agent n'est pas obligé d'inclure à l'agent tous les composants dont il peut avoir besoin par la suite. Chaque agent peut utiliser des composants qui faisaient auparavant partie d'autres agents. Cette approche est donc intéressante lorsque les adaptations requises pour un agent ne peuvent être anticipées, comme c'est le cas dans les systèmes ouverts.

MaDcAr-Agent respecte les propriétés des agents, citées au début de ce chapitre. La *capacité sociale* de l'agent est utilisée pour l'échange de composants comme indiqué dans le paragraphe précédent. L'agent reste complète-

ment *autonome* concernant l'adaptation puisqu'il incorpore un moteur MadCAr et peut décider de se réassembler sans l'intervention d'autres agents ou d'êtres humains. La *réactivité* et la *proactivité* sont obtenues respectivement par l'utilisation du contexte externe et du contexte interne pour décider - selon une politique d'adaptation - si l'agent doit adapter l'assemblage de composants de son niveau de base.

En résumé, MadCAr-Agent permet la construction d'agents avec des capacités d'adaptation et en encourageant la séparation des préoccupations tels que le comportement applicatif, le transfert d'état, la consommation de ressources matérielles, etc. Ces différents aspects sont pris en compte dans le respect des propriétés spécifiques aux agents et notamment l'autonomie.

Troisième partie

Réalisations

Chapitre 6

Expérimentations

Ce chapitre décrit les expérimentations que nous avons menées pour valider notre approche. Il contient plusieurs exemples qui illustrent les contributions des chapitres 4 et 5. La section 6.1 décrit le framework développé pour MaDcAr et MaDcAr-Agent. La section 6.2 présente un premier exemple d'application qui a été implémenté pour étudier la faisabilité de notre approche (rôles, configurations et résolution à base de contraintes dans une version simplifiée). La section 6.3 présente un second exemple d'application permettant d'illustrer les autres aspects de l'approche (déclenchement automatique, politique d'assemblage et transfert d'état) ; La section 6.4 présente un scénario applicatif d'agents auto-adaptables représentant des robots mobiles afin d'illustrer le modèle MaDcAr-Agent. Ce scénario s'inscrit dans le cadre du Projet AROUND [AP] qui concerne la conception et l'implémentation d'un système d'aide à la décision en temps-réel pour le redressement de désastres en zones urbaines. Enfin, la section 6.5 conclut le chapitre.

6.1 Framework développé pour MaDcAr et MaDcAr-Agent

Nous avons développé un framework pour la mise en œuvre de MaDcAr et de MaDcAr-Agent. Il est implémenté en *Smalltalk*, qui est un langage objet dynamique qui possède un environnement intégré facilitant le prototypage rapide. Nous utilisons plus particulièrement le dialecte libre *Squeak* [BDNP07]. Squeak est caractérisé par sa très haute portabilité permettant d'exécuter à l'identique des programmes sur une grande variété de systèmes (notamment Windows, Mac et Unix) et de machines (PC de bureau, PC portable, PDA, robot, console de jeu portable, ...). Par conséquent, nous pourrons mener des expérimentations quel que soit le scénario applicatif

choisi.

6.1.1 Architecture générale du framework

L'approche par framework signifie que notre implémentation est conçue de manière réutilisable en exprimant un ensemble de classes abstraites et la façon dont leurs instances interagissent [JF88]. En particulier, nous pouvons projeter le modèle MaDcAr sur différents modèles de composants. Pour prendre en charge un nouveau modèle de composants, il faut implanter un petit ensemble d'opérations typiques sur les composants (connexion/déconnexion de deux composants, activation/désactivation d'un composant et importation/exportation de l'état d'un composant). Le framework comporte plusieurs parties qui sont généralement représentées par un paquetage. Nous pouvons le décomposer de la manière suivante :

- paquetage `BackTalk` : il représente la librairie que nous utilisons pour la résolution de contraintes. La version originale a été conçue et implémentée en VisualWorks - un autre dialecte de Smalltalk - par Pierre Roy et François Pachet [RP97]. Dans un premier temps, nous avons effectué le portage partiel du code du solveur entre une vieille version VisualWorks et une version récente de Squeak. C'était suffisant pour nos premières expérimentations (voir la section 6.2), mais de nombreux bogues nous ont poussé à développer une nouvelle version appelée `BackTalkNG` [1]. Cette nouvelle version est plus restreinte (une dizaine de classes au lieu d'une centaine) mais aussi plus fiable. Nous l'avons utilisée dans le cadre des expérimentations présentées en section 6.4 ;
- paquetage `MaDcAr` : il contient les classes de base du modèle MaDcAr (rôle, connexion entre rôles, configuration, composant, moteur d'assemblage et application). Pour nos expérimentations, nous avons implémenté une spécialisation de MaDcAr pour le modèle de composants Fractal [BCS02]. L'outil résultant s'appelle *AutoFractal* [2]. C'est ce que nous utilisons pour l'exemple de la section 6.2 ;
- paquetage `FracTalk` : il s'agit de l'implantation en Smalltalk du modèle de composants Fractal. FracTalk [3] a été développé dans notre équipe par Houssam Fakih dans le cadre d'une thèse précédente sur les composants et les aspects ;
- paquetage `MaDcArAgent` : il s'agit d'une extension du paquetage MaDcAr permettant de concevoir des agents selon le modèle MaDcArAgent. En particulier, on y trouve les classes permettant à des agents

1. *BackTalk New Generation*, `http://www.squeaksource.com/BackTalk.html`
2. `http://vst.ensm-douai.fr/AutoFractal`
3. `http://vst.ensm-douai.fr/FracTalk`

de gérer leur contenu, en particulier grâce à un protocole d'échange de composants entre agents. Ce paquetage est en cours d'implémentation, c'est pourquoi nous ne détaillons pas l'aspect « gestion du contenu » dans l'exemple que nous avons développé (voir la section 6.4).

6.1.2 Création d'une application à l'aide du framework

Les étapes de création d'une application (classique ou agent) sont les suivantes :

- Création d'une interface homme-machine (si nécessaire) ;
- Création d'un composant infrastructure (si nécessaire) ;
- Spécification du contexte ;
- Spécification des politiques d'adaptation (assemblage et contenu) ;
- Spécification des rôles ;
- Spécification des configurations ;
- Spécification du réseau de transfert d'état ;
- Instantiation d'un moteur d'assemblage ;
- Choix de l'ensemble des composants utilisables par l'application.

Concrètement, chaque application hérite de la classe abstraite `MaDcArAbstractApplication`. Cette classe définit un ensemble des méthodes abstraites à implémenter pour chaque application (`createConfigurations`, `createStateTransferNet`, etc.). Dans les prochaines sections se trouvent les exemples de code pour ces méthodes.

6.2 Exemple de l'horloge adaptable

Les éléments présentés dans cette section concernent le premier prototype que nous avons implémenté. Nous explicitons la conception d'une application simple avec MadcAr/AutoFractal. De plus, nous illustrons le processus de (ré)assemblage de MadcAr à base de résolution de contraintes.

Ce prototype correspond à une version du modèle MadcAr *moins évoluée* que celle du chapitre 4. En effet, les rôles et les configurations n'ont pas de propriétés extra-fonctionnelles (CPU, mémoire, etc.). De plus, la politique d'assemblage est définie de manière simpliste (*i.e.* proche de la définition d'un « problème de résolution de contraintes » et sans processus d'optimisation de la décision des adaptations en fonction des propriétés extra-fonctionnelles) et le déclenchement des adaptations se fait de manière manuelle, grâce à une interface homme-machine (IHM). Enfin, le transfert d'état de l'application repose sur un mécanisme moins élaboré que le réseau de transfert d'état : il s'agit de définir une *configuration « pivot »*, puis de spécifier pour chaque

autre configuration une fonction d'importation de l'état depuis la configuration « pivot » et une fonction d'exportation de l'état vers la configuration « pivot ».

Ce simple prototype illustre comment un algorithme de résolution de contraintes permet de contrôler le réassemblage d'une application en fonction de sa spécification (rôles et configurations) sans faire de référence directe aux composants impliqués (découplage).

6.2.1 Description informelle

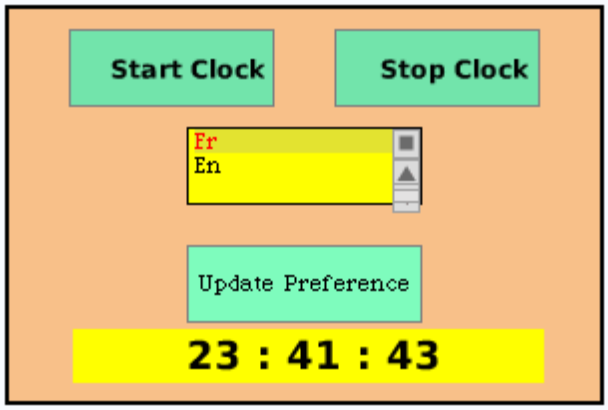

FIGURE 6.1 – IHM de l'application horloge adaptable.

L'exemple présenté ci-dessous est une application d'horloge. Dans cet exemple, c'est l'utilisateur de l'application qui est à l'origine du déclenchement du processus de réassemblage grâce à l'IHM (figure 6.1). Concrètement, l'IHM permet de mettre à jour ses préférences applicatives comme le choix du mode d'affichage et l'attribution d'horaires pour l'alarme.

6.2.2 Configurations

L'ensemble des configurations décrivant l'horloge est représenté dans la figure 6.2. Nous distinguons quatre configurations, dont deux permettent d'avoir une fonctionnalité d'alarme.

La configuration `configSimpleClock24` décrit une horloge comprenant quatre rôles. Chacun de ces rôles a comme multiplicités minimale et maximale la valeur 1. En d'autres termes, chaque rôle doit être rempli par un seul composant. Les rôles `Hours`, `Minutes` et `Seconds` représentent des compteurs circulaires décrivant une valeur qui évolue dans un intervalle. Les bornes d'un

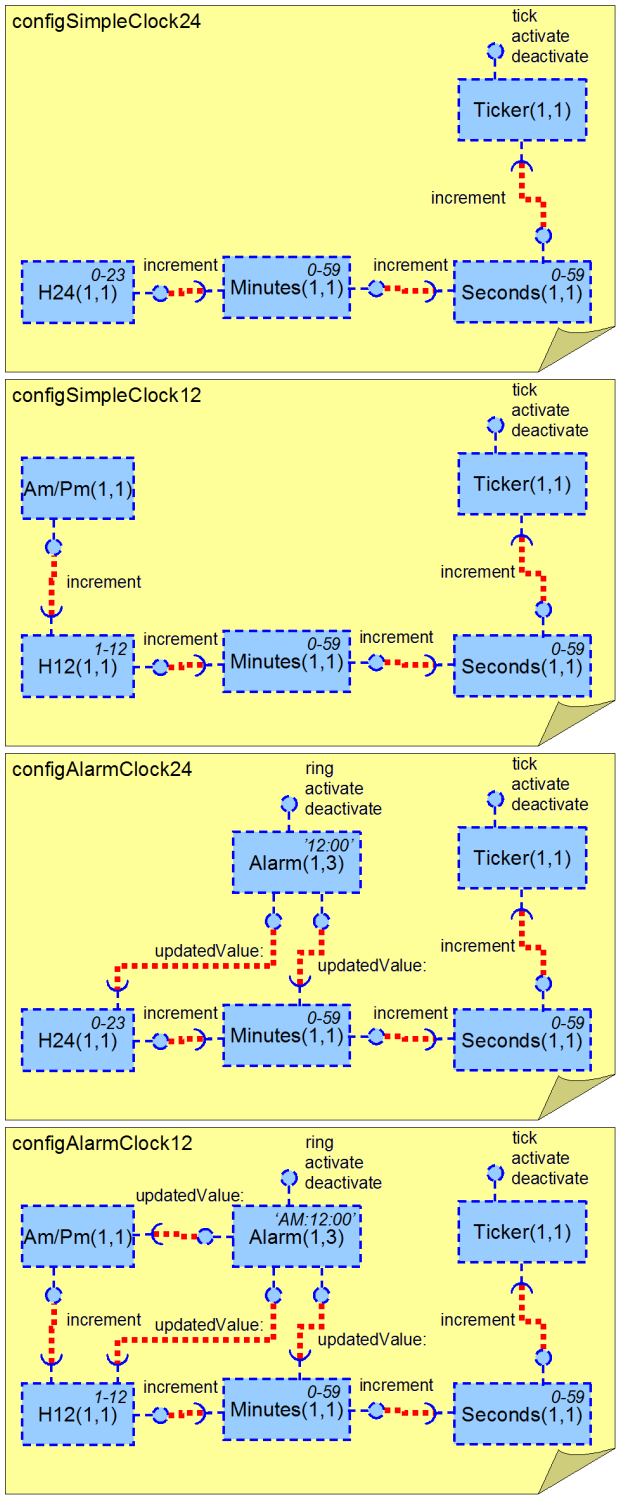

FIGURE 6.2 – Description de l'application horloge avec alarme.

```
roleTicker := Role named: 'roleTicker' min: 1 max: 1.
roleTicker addProvidedInterfaceNamed: #ticking
           selectors: { #tick. #activate. #deactivate }.
roleTicker addRequiredInterfaceNamed: #tick
           selectors: { #increment }.

roleSeconds := Role named: 'roleSeconds' min: 1 max: 1.
roleSeconds addProvidedInterfaceNamed: #modify
            selectors: { #increment }.
roleSeconds addRequiredInterfaceNamed: #notify
            selectors: { #increment }.
roleSeconds addAttributeNamed: #min value: 0.
roleSeconds addAttributeNamed: #max value: 59.
roleSeconds addAttributeNamed: #value initValue: 59.

roleMinutes := Role named: 'roleMinutes' min: 1 max: 1.
roleMinutes addProvidedInterfaceNamed: #modify
            selectors: { #increment }.
roleMinutes addRequiredInterfaceNamed: #notify
            selectors: { #increment }.
roleMinutes addAttributeNamed: #min value: 0.
roleMinutes addAttributeNamed: #max value: 59.
roleMinutes addAttributeNamed: #value initValue: 59.

roleHours24 := Role named: 'roleHours24' min: 1 max: 1.
roleHours24 addProvidedInterfaceNamed: #modify
            selectors: { #increment }.
roleHours24 addAttributeNamed: #min value: 0.
roleHours24 addAttributeNamed: #max value: 23.
roleHours24 addAttributeNamed: #value initValue: 23.
```

FIGURE 6.3 – Code des rôles de la configuration `configSimpleClock24`.

intervalle sont des attributs spécifiés (en haut du coin droit) dans chaque rôle de la figure 6.2. Le rôle `Hours` décrit un compteur qui tourne entre 0 et 23. Le rôle `Minutes` décrit un compteur qui tourne entre 0 et 59, et qui incrémente Hours à chaque tour. Le rôle `Seconds` décrit un compteur qui tourne entre 0 et 59, et qui incrémente Minutes à chaque tour. `Seconds` est incrémenté chaque seconde par un composant remplissant le rôle `Ticker`. Le rôle `Ticker` représente un composant actif, *i.e.* un composant doté d'un processus léger contrôlable à travers une interface [4], qui réalise la tâche d'incrémentation à chaque seconde. Le code de ces rôles est donné dans la figure 6.3. Le code des liens entre ces rôles est donné dans la figure 6.4.

```
configSimpleClock24 := Configuration named: 'configSimpleClock24'.
configSimpleClock24 addRoles: {
  roleTicker. roleSeconds. roleMinutes. roleHours24 }.
configSimpleClock24
  linkRole: roleTicker port: #tick
  withRole: roleSeconds port: #modify.
configSimpleClock24
  linkRole: roleSeconds port: #notify
  withRole: roleMinutes port: #modify.
configSimpleClock24
  linkRole: roleMinutes port: #notify
  withRole: roleHours24 port: #modify.
```

FIGURE 6.4 – Code de la configuration `configSimpleClock24`.

La configuration `configSimpleClock12` diffère de la configuration `configSimpleClock24` sur deux points. Son rôle `Hours` décrit un compteur qui boucle entre 1 et 12. De plus, il possède un rôle supplémentaire `Am/Pm` qui permet à un composant de changer la valeur de son attribut de AM à PM et *vice versa* suivant le moment de la journée.

Les configurations `configAlarmClock24` et `configAlarmClock12` sont respectivement similaires aux configurations `configSimpleClock24` et `configSimpleClock12`, mais elles permettent d'avoir en plus une fonctionnalité d'alarme. Pour ce faire, elles introduisent chacune un rôle `Alarm` qui permet de définir jusqu'à trois horaires pour la sonnerie, d'où la multiplicité maximale de 3. La figure 6.5 montre le code du rôle `Alarm` de la configuration `configAlarmClock12`. Les interfaces `#evalMinutes`, `#evalHours` et `#evalAmPm` permettent d'accéder à l'heure courante et de la comparer à l'heure de sonnerie spécifiée dans l'attribut `#alarmTime`.

4. Conformément aux hypothèses de travail énoncés dans la section 4.2, un composant actif se caractérise par la présence d'une interface qui fournit les opérations activation et désactivation.

```
roleAlarm := Role named: 'roleAlarm' min: 1 max: 3.
roleAlarm addProvidedInterfaceNamed: #evalMinutes
          selectors: { #updatedValue: }.
roleAlarm addProvidedInterfaceNamed: #evalHours
          selectors: { #updatedValue: }.
roleAlarm addProvidedInterfaceNamed: #evalAmPm
          selectors: { #updatedValue: }.
roleAlarm addProvidedInterfaceNamed: #ringing
          selectors: { #ring. #activate. #deactivate }.
roleAlarm addAttributeNamed: #alarmTime value: 'AM:12:00'.
```

FIGURE 6.5 – Code du rôle Alarm de la configuration `configAlarmClock12`.

6.2.3 Contexte

```
displayPreferenceSensor := Sensor new.
displayPreferenceSensor resource: 'external/software/IHM/displayModeButton'.
displayPreferenceSensor operation: #setDisplayMode call: #onClick.
selectedDisplayPreferenceSensor := Sensor new.
selectedDisplayPreferenceSensor resource: 'internal/software/IHM/displayMode'.
selectedDisplayPreferenceSensor operation: #hasChanged.
selectedDisplayPreferenceSensor operation: #isSetTo24HoursTimeSystem.
alarmDefinitionSensor := Sensor new.
alarmDefinitionSensor resource: 'external/software/IHM/alarmButton'.
alarmDefinitionSensor operation: #setAlarms call: #onClick.
definedAlarmsSensor := Sensor new.
definedAlarmsSensor resource: 'internal/software/IHM/alarmTimes'.
definedAlarmsSensor operation: #hasChanged.
definedAlarmsSensor operation: #number.
. . .
definedAlarmsSensor operation: #getThirdAlarmTime.
```

FIGURE 6.6 – Code du contexte de l'application horloge.

La figure 6.6 définit le contexte pour cette application. Le contexte contient plusieurs capteurs. Deux des capteurs sont dédiés au déclenchement des adaptations. Le capteur `displayPreferenceSensor` est déclenché lorsqu'on clique sur le bouton de modification du format d'affichage de l'heure et le capteur `alarmDefinitionSensor` est déclenché lorsqu'on valide l'ajout ou la suppression d'une valeur d'alarme.

6.2.4 Politique d'assemblage

Le code de la pseudo-politique d'assemblage de l'horloge est représenté par la figure 6.7. Ce code joue le rôle de « fonction de sélection » (au sens de

```
constraintSet reinitialize.
(selectedDisplayPreferenceSensor isSetTo24HoursTimeSystem)
  ifTrue: [
    constraintSet add:
      (newConfig doesNotIncludes: 'roleAmPm')]
  ifFalse: [
    constraintSet add:
      (newConfig includes: 'roleAmPm')].
(definedAlarmsSensor number = 0)
  ifTrue: [
    constraintSet add:
      (newConfig doesNotIncludes: 'roleAlarm')]
  ifFalse: [
    constraintSet add:
      (newConfig includes: 'roleAlarm').
    constraintSet add:
      ((newConfig roleNamed: 'roleAlarm') multiplicity
        = (definedAlarmsSensor number)).].
```

FIGURE 6.7 – Code de la politique d'assemblage de l'application horloge.

la sous-section 4.2.5), mais dans une version *ad hoc* et simpliste (pas d'utilisation de fonctions de caractérisation et pas de processus d'optimisation de la décision du réassemblage).

La variable `constraintSet` contient l'ensemble de contraintes représentant le problème d'assemblage. En particulier, les spécifications des configurations y sont injectées en tant que contraintes. A chaque fois que la politique d'assemblage est exécutée, cette variable est réinitialisée d'après les configurations et les composants disponibles. Ensuite, en fonction de l'état du contexte (préférence d'affichage et alarmes définies), on ajoute des contraintes supplémentaires qui permettent de restreindre le choix pour la nouvelle configuration.

6.2.5 Transfert d'état

Concernant le transfert d'état, nous avons défini l'état de l'application à travers une configuration « pivot » (`configSecondsPerDay`), dont le code est donné dans la figure 6.8. Cette configuration contient un rôle `roleSecondsPerDay` doté d'un attribut entier qui représente l'heure courante en secondes et un rôle `roleAlarm` doté d'un tableau de taille 3 contenant l'heure des alarmes.

Les méthodes d'importation et d'exportation de l'état de la configuration `configSimpleClock24` vis-à-vis de la configuration « pivot » sont données dans la figure 6.9. La méthode `chosenComponent` permet de désigner

```
roleSecondsPerDay := Role named: 'roleSecondsPerDay' min: 1 max: 1.
roleSecondsPerDay addAttributeNamed: #timeInSeconds initValue: 0.

roleAlarm := Role named: 'roleAlarm' min: 1 max: 3.
roleAlarm addAttributeNamed: #alarmTimes value: { ''. ''. '' }.

configSecondsPerDay := Configuration named: 'configSecondsPerDay'.
configSecondsPerDay addRole: roleSecondsPerDay.
configSecondsPerDay addRole: roleAlarm.
```

FIGURE 6.8 – Code de la configuration `configSecondsPerDay`.

```
export
  | h m s state |
  h := (self roles at: 'roleHours24') chosenComponent
    getAttribute: #value.
  m := (self roles at: 'roleMinutes') chosenComponent
    getAttribute: #value.
  s := (self roles at: 'roleSeconds') chosenComponent
    getAttribute: #value.
  state := Dictionary new.
  state at: #timeInSeconds put: ((h*60*60) + (m*60) + s).
  ^ state.

  | time h m s |
  time := state at: #timeInSeconds.
                ifAbsent: [ ^ self ].
  h := (time // (60*60)) \\ 24.
  (self roles at: 'roleHours24') chosenComponent
    setAttribute: #value to: h.
  m := (time - (h*60*60)) // 60).
  (self roles at: 'roleMinutes') chosenComponent
    setAttribute: #value to: m.
  s := time \\ 60).
  (self roles at: 'roleSeconds') chosenComponent
    setAttribute: #value to: s.
```

FIGURE 6.9 – Code pour le transfert d'état de la configuration `configSimpleClock24`.

le composant qui remplit un rôle donné. Les méthodes `getAttribute:` et `setAttribute:to:` sont définies dans le package `MaDcAr` et elles sont implémentées dans le package spécifique à un modèle de composant (par exemple `Fractal`).

6.2.6 (Ré)assemblage de l'horloge

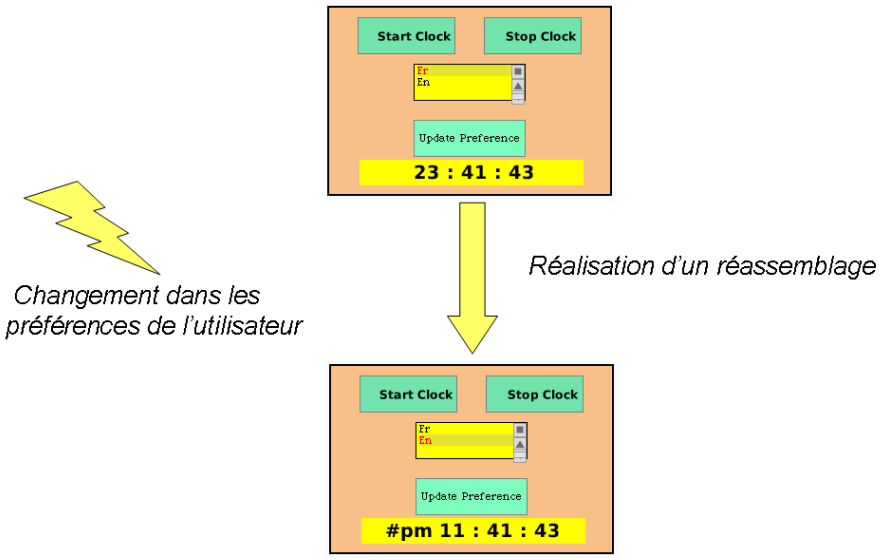

FIGURE 6.10 – Cas de réassemblage de l'horloge.

La figure 6.10 illustre le cas de réassemblage que nous allons expliciter. Il s'agit de passer d'un affichage à la française (*i.e.* sur 24 heures) vers un affichage de type anglo-saxon (*i.e.* avec les mentions AM ou PM). La configuration courante est `configSimpleClock24` puisqu'aucune alarme n'est supposée être réglée. Nous supposons que l'application contient six composants : « am/pm », « tic-tac », trois compteurs circulaires et deux composants d'alarme.

6.2.6.1 Décision du réassemblage

L'exécution du solveur de contraintes permet d'avoir une *matrice de compatibilité* entre les rôles et les composants, comme le montre la table 6.1. Dans cette table, chaque configuration est représentée par l'ensemble de ses rôles, chaque rôle étant sur une ligne différente. On peut noter que certains composants comme « tic-tac » peuvent remplir plusieurs rôles issus de configurations

différentes. De même, certains composants comme « circular counter 1 » sont compatibles avec plusieurs rôles au sein même d'une configuration.

	Rôles / Composants	tic-tac	circular Counter1	circular Counter2	circular Counter3	circular Counter4	alarm1	alarm2	am/pm
configSimpleClock24	Ticker(1,1)	X							
	Seconds(1,1)		X	X	X	X			
	Minutes(1,1)		X	X	X	X			
	Hours24(1,1)			X	X	X			
configSimpleClock12	Ticker(1,1)	X							
	Seconds(1,1)		X	X	X	X			
	Minutes(1,1)		X	X	X	X			
	Hours12(1,1)			X	X	X			
	Am/Pm(1,1)								X
configAlarmClock24	Ticker(1,1)	X							
	Seconds(1,1)		X	X	X	X			
	Minutes(1,1)		X	X	X	X			
	Hours24(1,1)			X	X	X			
	Alarm(1,3)						X	X	
configAlarmClock12	Ticker(1,1)	X							
	Seconds(1,1)		X	X	X	X			
	Minutes(1,1)		X	X	X	X			
	Hours12(1,1)			X	X	X			
	Am/Pm(1,1)								X
	Alarm(1,3)						X	X	

TABLE 6.1 – Exemple de matrice de compatibilité entre des rôles et des composants.

Le contexte (*i.e.* les préférences de l'utilisateur de l'horloge) et la politique d'assemblage conduisent finalement à la sélection de la configuration `configAlarmClock12` :

- le test `selectedDisplayPreferenceSensor isSetTo24HoursTimeSystem` renvoie la valeur faux, puisque par hypothèse l'utilisateur vient de mettre à jour la préférence d'affichage, donc la nouvelle configuration doit contenir le rôle `Am/Pm` ;
- le test `definedAlarmsSensor number = 0` renvoie la valeur vrai, car sinon `configAlarmClock12` n'aurait pas pu être la configuration précédente, donc la nouvelle configuration ne doit pas contenir le rôle `Alarm` ;

`configAlarmClock12` peut être remplie par l'ensemble de composants suivant : *tic-tac, circular counter 1, circular counter 2, circular counter 4, et am/m.*

6.2.6.2 Réalisation du réassemblage

La trace du réassemblage de `configSimpleClock24` vers `configSimpleClock24` est donnée par la figure 6.11. Toutes connexions entre composants de l'assemblage courant sont supprimées à chaque réassemblage. Puis, le transfert d'état s'effectue entre l'ancienne configuration `configSimpleClock24` et la nouvelle configuration `configAlarmClock12`

```
Components deactivation
Unbinding cc4#notify from cc3#modify
Unbinded cc2#notify from cc4#modify
Export  h:23 m:41 s:43 ToState:85303
Import c:pm h:11 hc:12 m:41 s:43 FromState:85303
Binding of cc1#notify to amPmSwitch#modify
Binding of cc4#notify to cc1#modify
Binding of cc2#notify to cc4#modify
Binding of tictac#tick to cc2#modify
Components reactivation
```

FIGURE 6.11 – Trace d'un réassemblage de l'application horloge.

en passant par la configuration pivot. Ensuite, le nouvel assemblage est construit en fonction des composants choisis précédemment.

6.2.7 Discussion

Cet exemple montre comment un algorithme de résolution de contraintes permet de contrôler le réassemblage d'une application en fonction de sa spécification (rôles et configurations) sans faire de référence directe aux composants impliqués (découplage). Cela illustre un fort potentiel de *réutilisabilité* dans notre modèle puisque non seulement les configurations d'une application sont utilisables avec différents jeux de composants mais aussi le mécanisme de (ré)assemblage est complètement générique grâce à l'utilisation d'un solveur de contraintes. La programmation par contraintes permet au concepteur de se concentrer sur la description du problème de réassemblage plutôt que sur sa résolution.

Concernant la politique d'assemblage, l'intérêt d'utiliser des contraintes est qu'il n'est pas nécessaire de décrire exactement tout ce qu'il faut réaliser comme adaptation dans chaque situation contextuelle possible. Par exemple, dans la figure 6.7, tous les rôles que la nouvelle configuration doit contenir ne sont pas énumérés.

6.3 Exemple du client de discussion adaptatif

Cette section illustre plus particulièrement la définition de contexte, de politique d'assemblage et de transfert d'état dans MADCAR.

6.3.1 Description informelle

Considérons une application de discussion dans laquelle plusieurs participants distants peuvent s'échanger des messages via un forum. Ce forum diffuse chaque message qu'il reçoit à l'ensemble des participants connectés. Pour illustrer le principe de ré-assemblage dynamique et automatique avec MADCAR, nous nous intéressons à la conception d'un logiciel « client de discussion » qui permet aux participants d'envoyer des messages malgré les éventuelles déconnexions. Les messages édités hors-ligne sont stockés dans un composant tampon (*buffer*) et puis ré-expédiés vers le forum lorsque la connexion réseau est rétablie.

6.3.2 Configurations

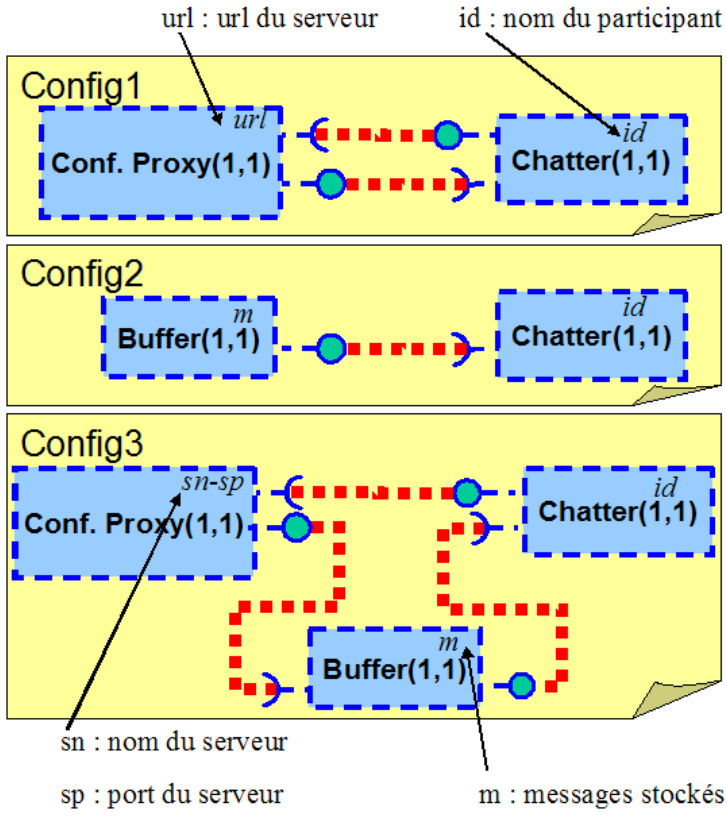

FIGURE 6.12 – Configurations du client de discussion.

La description de l'application « client de discussion » contient trois configurations : `Config1`, `Config2` and `Config3` (voir figure 6.12). Suppo-

sons qu'au démarrage de l'application, le réseau est disponible. Le moteur MadCar assemble alors les composants du client de discussion d'après la configuration `Config1`. Ainsi, le participant (rôle `Chatter`) est connecté directement au proxy du forum de discussion (rôle `ConferenceProxy`). Lorsque la connexion réseau est perdue, le moteur d'assemblage réassemble l'application d'après `Config2`. Ainsi, les messages envoyés par le participant sont stockés dans un composant tampon. Lorsque la connexion réseau est rétablie, l'application est réassemblée selon `Config3`, de manière à permettre l'envoi des messages en attente. Finalement, l'application peut revenir à sa configuration de départ `Config1` lorsque le composant qui joue le rôle « Buffer » est totalement vidé.

```
"---------- Pertinence ----------"
cfRelevance := [ :context |
  (context at: #ethernetNetworkSensor) isAvailable
  | (context at: #wifiNetworkSensor) isAvailable)
    ifFalse: [ 0 ]
    ifTrue: [
      ((context at: #bufferSizeSensor)] isEmpty)
        ifTrue: [ 100 ]
        ifFalse: [ 50 ]
      ]
  ].
"----------- CPU -----------"
cfCPU := [:components :context |
 | requiredCPU availableCPU |
  requiredCPU := components inject: 0 into:
    [ :sum :each | (each contract: #maxCPU) + sum ].
  availableCPU := (context at: #cpuSensor) getAmountOfMaxCPU.
  (availableCPU - requiredCPU <= 0)
    ifTrue: [ 0 ]
    ifFalse: [ 100 * requiredCPU / availableCPU ]
  ].
"----------- Mémoire -----------"
. . .
"----------- Energie -----------"
. . .
```

Figure 6.13 – Code des fonctions de caractérisation de `Config1`.

Des exemples de code pour spécifier les fonctions de caractérisation de ces configurations sont donnés par les figures 6.13, 6.14 et 6.15. Ces fonctions utilisent les données du contexte défini dans la figure 6.16. Elles sont utilisées dans la politique d'assemblage pour sélectionner la configuration et les composants à utiliser pour l'assemblage étant donné le contexte courant.
Concernant la pertinence :

```
"---------- Pertinence ----------"
cfRelevance := [ :context |
  (context at: #ethernetNetworkSensor) isAvailable
  | (context at: #wifiNetworkSensor) isAvailable)
    ifFalse: [ 100 ]
    ifTrue: [ 10 ]
  ].
"----------- CPU -----------"
cfCPU := [:components :context |
 | requiredCPU availableCPU |
  requiredCPU := components inject: 0 into:
    [ :sum :each | (each contract: #maxCPU) + sum ].
  availableCPU := (context at: #cpuSensor) getAmountOfMaxCPU.
  (availableCPU - requiredCPU <= 0)
    ifTrue: [ 0 ]
    ifFalse: [ 100 * requiredCPU / availableCPU ]
  ].
"----------- Mémoire -----------"
. . .
"----------- Energie -----------"
. . .
```

FIGURE 6.14 – Code des fonctions de caractérisation de `Config2`.

- La pertinence de `Config1` est 0 lorsqu'il n'y a pas de réseau, 50 lorsque le réseau est disponible et le tampon de messages non délivrés est n'est pas vide, et 100 lorsque le réseau est disponible et le tampon est vide.
- `Config2` est totalement pertinent (*i.e.* valeur 100) lorsqu'il n'y a pas de réseau, et peu pertinent sinon (*i.e.* valeur 10).
- La pertinence de `Config3` est 0 lorsqu'il n'y a pas de réseau, 100 lorsque le réseau est disponible et le tampon de messages non délivrés est n'est pas vide, et 50 lorsque le réseau est disponible et le tampon est vide.
- Concernant les autres propriétés des configurations, notamment le besoin en CPU, la fonction caractérisation est un ratio entre la quantité disponible (issus du contexte) et la quantité requise par les composants à utiliser.

Notons que pour définir des fonctions en Smalltalk, nous utilisons des objets appelés « blocs ». Par exemple, le bloc « `[ :x :y | x + y ]` » s'apparente à la lambda-expression $\lambda xy.(x+y)$, c'est-à-dire la fonction addition. Ce bloc peut s'évaluer grâce à la méthode `value:value:`. Ainsi, le code « `[:x :y | x + y] value: 2 value: 3` » renvoie la valeur 5.

```
"---------- Pertinence ----------"
cfRelevance := [ :context |
  (context at: #ethernetNetworkSensor) isAvailable
  | (context at: #wifiNetworkSensor) isAvailable)
    ifFalse: [ 0 ]
    ifTrue: [
      ((context at: #bufferSizeSensor)] isEmpty)
        ifTrue: [ 10 ]
        ifFalse: [ 100 ]
      ]
  ].
"----------- CPU -----------"
cfCPU := [:components :context |
 | requiredCPU availableCPU |
  requiredCPU := components inject: 0 into:
    [ :sum :each | (each contract: #maxCPU) + sum ].
  availableCPU := (context at: #cpuSensor) getAmountOfMaxCPU.
  (availableCPU - requiredCPU <= 0)
    ifTrue: [ 0 ]
    ifFalse: [ 100 * (available - requiredCPU) / availableCPU ]
  ].
"----------- Mémoire -----------"
. . .
"----------- Energie -----------"
. . .
```

FIGURE 6.15 – Code des fonctions de caractérisation de `Config3`.

```
ethernetNetworkSensor := Sensor new.
ethernetNetworkSensor resource: 'external/hardware/network/ethernet'.
ethernetNetworkSensor operation: #isAvailable updatePeriod: 1000.
wifiNetworkSensor := Sensor new.
wifiNetworkSensor resource: 'external/hardware/network/wifi'.
wifiNetworkSensor operation: #isAvailable updatePeriod: 1000.
bufferSizeSensor := Sensor new.
bufferSizeSensor resource: 'internal/stateTransferNet/bufferNode'.
bufferSizeSensor operation: #isEmpty updatePeriod: 500.
bufferSizeSensor operation: #getCurrentSize.
. . .
cpuSensor := Sensor new.
cpuSensor resource: 'external/hardware/system/cpu'.
cpuSensor operation: #getAmountOfAvailableCPU updatePeriod: 1000.
cpuSensor operation: #atLeastTenPercentOfAvailableCPU updatePeriod: 1000.
cpuSensor operation: #getAmountOfMaxCPU.
```

FIGURE 6.16 – Code du contexte du client de discussion.

6.3.3 Contexte

La figure 6.16 fournit le code du contexte pour le client de discussion. Dans cet exemple, un réassemblage est déclenché soit lorsque le statut de la connexion réseau change, soit lorsque le tampon stockant des messages non expédiés devient vide. Par conséquent, nous avons besoin de plusieurs sondes : `ethernetNetworkSensor` surveille le réseau ethernet, `wifiNetworkSensor` surveille le réseau WiFi et `bufferSizeSensor` surveille l'attribut variable tampon utilisée pour les messages. Chaque sonde se caractérise par une ressource [5] observée ainsi que les différentes opérations sur la ressource pouvant être utilisée pour spécifier la politique d'assemblage. Ces sondes sont dîtes « actives » parce qu'un processus léger est utilisé pour exécuter périodiquement certaines opérations sur les ressources. C'est pourquoi nous spécifions un délai à respecter entre deux exécutions. Hormis ces sondes spécifiques à l'application, le contexte doit contenir des sondes qui concernent les ressources matérielles (CPU, mémoire et énergie) : `cpuSensor`, ...

6.3.4 Politique d'assemblage

La politique d'assemblage du client de discussion est donnée par la figure 6.17. Elle contient une liste de situations contextuelles susceptibles de déclencher un réassemblage de l'application. En fait, il y a un processus de notre framework qui évalue régulièrement [6] chacune de ces situations contextuelles. Si l'une d'elles correspond au contexte courant, alors le processus d'assemblage choisit une configuration et des composants d'après la fonction de sélection `configAndComponentSelectingFunction` [7].

Notons que si le moteur d'assemblage ne dispose que très peu de ressources matérielles, il est préférable de n'autoriser le déclenchement que lors d'un changement important du contexte : par exemple, lorsqu'une des situations contextuelles qui n'étaient pas vraies pour l'évaluation précédente devient valide. Pour ce faire, il suffit de modifier le gestionnaire de contexte pour mémoriser dans un tableau le booléen correspondant à l'évaluation de chaque élément de `contextualSituations`.

5. Une ressource est symbolisée par un chemin à la manière du gestionnaire de contexte WildCAT [DL05].

6. La détermination des « périodes de d'adaptation » est explicitée dans les sous-sections 4.2.5 et 4.2.6.

7. Elle était notée SF_{bcc} dans le chapitre 4.2.

```
"---------- Déclenchement ----------"
minDurationBetweenAdaptationTriggering := 2000.
contextualSituations := {
  (ethernetNetworkSensor isAvailable) & (bufferSizeSensor isEmpty).
  (ethernetNetworkSensor isAvailable) & (bufferSizeSensor isEmpty not).
  (ethernetNetworkSensor isAvailable not).
  (wifiNetworkSensor isAvailable) & (bufferSizeSensor isEmpty).
  . . .
  (cpuSensor getAmountOfAvailableCPU > 1000)
  & (memorySensor getAmountOfAvailableMemory >= 512).
  (cpuSensor getAmountOfAvailableCPU < 200).
  (memorySensor getAmountOfAvailableMemory < 50).
  (energySensor getAmountOfAvailableEnergy < 10).
   }.
"----------- Décision -----------"
maxDurationForAdaptationDecision := 1000.
configAndComponentSelectingFunction := [ :config :components :context |
    (40 * config cfRelevance value: context)
  - (20 * config cfEnergy value: components value: context)
  - (10 * config cfMemory value: components value: context)
  - (10 * config cfCPU value: components value: context)
  ].
```

FIGURE 6.17 – Code de la politique d'assemblage du client de discussion.

6.3.5 Transfert d'état

L'application de client de discussion que nous avons décrite est constituée de trois configurations. Chacune de ces configurations comporte quelques attributs. Pour maintenir automatiquement un état cohérent dans l'application, il faut d'abord spécifier quels sont les attributs variables et quels sont les attributs fixes. Ensuite, il faut spécifier les modalités de transfert d'état grâce au réseau de transfert d'état.

6.3.5.1 Identification de l'état

La description de l'application spécifie 5 attributs variables différents :

1. *id* correspond au nom ou identifiant du participant. Il apparaît dans les rôles `Chatter` des trois configurations.
2. *m* correspond à une liste de messages à expédier. Il apparaît dans les rôles `Buffer` des configurations `Config2` et `Config3`.
3. *url* correspond à une URL désignant le forum de discussion distant. Il apparaît dans le rôle `Conference Proxy` dans la configuration `Config1`.
4. *sn* (server name) correspond au nom du serveur du forum distant. Il apparaît dans le rôle `Conference Proxy` dans la configuration `Config3`.

5. *sp* (server port) correspond au numéro de port du serveur de forum distant. Il apparaît dans le rôle `Conference Proxy` dans la configuration `Config3`.

6.3.5.2 Mise en œuvre du transfert d'état

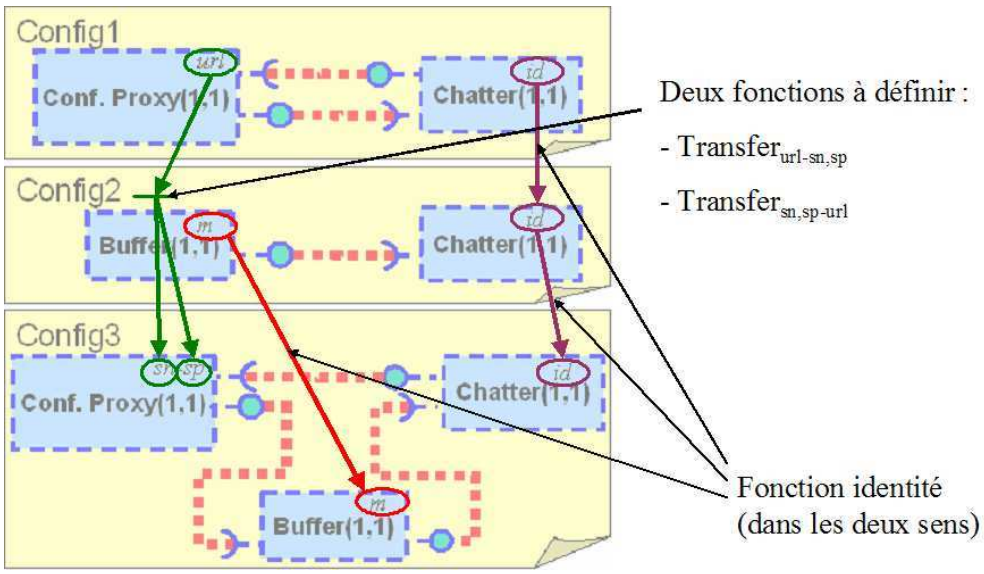

FIGURE 6.18 – Réseau de transfert d'état du client de discussion.

Le réseau de transfert d'état de cette application est constitué de trois composantes connexes, comme on peut le voir sur la figure 6.18. Pour les attributs *id*, étant identiques dans les trois configurations, ils sont reliés par la fonction identités. Idem pour les attributs *m* qui correspondent à la liste des messages en attente dans les tampons. Par contre, nous devons définir les fonctions pour lier les attributs des rôles `Conference Proxy` comme suit (en pseudo code) :

- $Transfer_{sn,sp-url}$: `url := sn + ':' + (sp.toString)` .
- $Transfer_{url-sn,sp}$: `sn := url.cutBeforeLast(':')` et
  `sp := url.cutAfterLast(':').toInteger` .

Comme les différentes fonctions utilisées dans le réseau de transfert d'état n'introduisent aucune perte d'information, le choix des attributs maîtres peut se faire de manière arbitraire. Grâce à ce réseau, le moteur d'assemblage peut réaliser automatiquement les transferts d'état à chaque ré-assemblage.

6.3.6 Discussion

Cet exemple permet d'illustrer plusieurs aspects de MaDcAr qui ont été simplifiés dans l'exemple de la section précédente : transfert d'état et décision d'assemblage. Ci-dessous, nous montrons que notre approche permet de séparer les préoccupations mais d'une manière différente de la programmation par aspects [KLM+97].

Nous avons utilisé un réseau de transfert d'état plutôt qu'une configuration pivot pour avoir un meilleur *découplage des configurations par rapport à l'aspect « description d'assemblages » et l'aspect « transfert d'état »*.
Nous pouvons évaluer l'intérêt de cette évolution en estimant la complexité de l'ajout et de la suppression d'une configuration en comptant le nombre de fonctions de transfert à définir par l'administrateur de l'application.
Soit c le nombre de configurations d'une application et a le nombre moyen d'attributs par configuration.

- Avec une configuration pivot, l'ajout ou la suppression d'une configuration $Config$ est potentiellement complexe car elle peut induire de mettre à jour la configuration pivot si certains des attributs de $Config$ sont sémantiquement plus riches que ceux de la configuration pivot [8]. La modification de la configuration pivot implique de mettre à jour les méthodes de transfert d'état de l'ensemble des configurations. Donc, la complexité de l'ajout ou de la suppression d'une configuration est en $O(a)$ si la configuration pivot ne change pas et elle est en $O(a * c + a)$ sinon. En outre, plus on ajoute de configurations ayant des attributs spécifiques, plus les méthodes de transfert d'état entre la configuration pivot et chaque autre configuration deviennent complexes, sans compter que la configuration pivot risque de devenir incompréhensible.
- Avec un réseau de transfert d'état, l'ajout ou la suppression d'une configuration implique une complexité raisonnable aussi pour l'administrateur de l'application. La mise à jour du réseau de transfert d'état par l'administrateur après l'ajout ou la suppression d'une configuration est de complexité polynomiale en fonction du nombre d'attributs de cette configuration (en $O(a)$), car chaque attribut possède au plus un attribut (ou groupe d'attributs) maître et un attribut (ou groupe d'attributs) esclave. En outre, même si l'état de l'application peut devenir globalement complexe, la plupart des fonctions de transfert à définir pour l'ajout d'une configuration peuvent se déduire en observant des configurations qui lui ressemblent.

Par ailleurs, nous avons doté les configurations de différentes fonctions

8. Dans l'exemple de l'horloge (section 6.2), il faut modifier la configuration pivot si on supprime les deux configurations qui impliquent la fonctionnalité d'alarme.

de caractérisation de leurs propriétés (pertinence, CPU, mémoire et énergie). Ces fonctions sont utilisées par la politique d'assemblage de l'application, en particulier à travers la fonction de sélection. Cela permet d'avoir un meilleur *découplage entre l'aspect « description d'assemblage » et l'aspect « décisions d'adaptation »*.
Dans l'exemple du client de discussion adaptatif, les pertinences des configurations sont facilement comparables entre elles car la valeur renvoyée par leur fonction de caractérisation est comprise entre 0 et 100. On peut remarquer que la modification d'une fonction de pertinence [9] ne nécessite pas forcément de modifier la fonction caractéristique de pertinence des configurations existantes. Par exemple, la configuration `Config3` pourrait avoir une pertinence de 100 dans tout les situations contextuelles car elle regroupe les fonctionnalités des deux autres configurations. Cela n'empêchera pas les configurations d'être sélectionnées dans la mesure où il existe des situations contextuelles pour lesquelles `Config1` et `Config2` sont meilleures, notamment parce que `Config3` sollicite plus de ressources matérielles.

L'exemple de cet section montre à quel point la modélisation d'une application par différentes configurations alternatives dans MaDcAr est en rupture avec l'approche classique consistant à vouloir concevoir une application comme doté d'une « configuration unique et parfaite ».

6.4 Exemple des robots auto-adaptables

Pour utiliser les concepts de MaDcAr-Agent, nous nous basons sur un scénario applicatif se déroulant dans le cadre de la robotique de sauvetage. Nous ne discutons pas du transfert d'état dans cet exemple.

6.4.1 Description informelle

Notre cas d'étude consiste à modéliser des robots qui doivent évoluer dans un environnement incertain, dans le cadre d'une mission de secours.

6.4.1.1 Projet AROUND

Le climat global change et provoque de plus en plus de catastrophes naturelles (tremblements de terre, typhons, ...). Les plus menacés par ces phénomènes météorologiques extrêmes sont les populations qui vivent dans les zones urbaines des pays en voie de développement, comme le Vietnam.

9. On peut faire un raisonnement analogue concernant l'ajout d'une nouvelle configuration.

Ces pays n'ont généralement pas accès à des systèmes pour l'acquisition de données en temps-réél (évaluation de dommages, localisation de victimes, identification des routes épargnées, ...) et la prise efficace de décision. Pourtant, « la valeur des technologies avancées dans la lutte contre ces désastres est largement reconnue » [Uni05]. Le *projet AROUND*[10] [AP] concerne la conception et l'implémentation d'un système d'aide à la décision en temps-réel pour le redressement de désastres en zones urbaines. Ce projet repose sur une combinaison entre des Systèmes d'Informations Géographiques (SIGs) globaux à différentes échelles et des capacités de détections automatiques locales. Il vise principalement les grandes villes asiatiques en cas de moussons ou d'autres catastrophes naturelles.

Dans le cadre du projet AROUND, notre équipe est impliquée dans le déploiement et la coordination d'une flotte de robots autonomes, bon marché et avec des capacités de communication. Ces robots doivent par exemple recueillir des informations utiles en temps réel grâce à leurs capteurs. Les robots sont en charge de planifier leurs chemins (selon leurs capacités physiques et leurs contexte environnant), en préservant leur autonomie énergétique, en se rechargeant de manière autonome et en interprétant les commandes provenant du centre de commandement des secours. Pour nos expérimentations, nous disposons de PDAs communiquants ainsi que de robots *Wifibots* [WB] qui sont équipés d'une caméra rotative, d'odomètres[11], de capteurs infrarouges et d'un dispositif de communication WiFi. En terme de capacités matérielles (CPU, mémoire et temps d'autonomie des batteries), un robot Wifibot est équivalent à un PDA.

6.4.1.2 Scénario et positionnement du problème

Dans ce cadre applicatif, nous ne présentons pas une mission de secours dans son ensemble, mais nous nous intéressons principalement à la tâche de localisation de chemins accessibles par exploration. Les robots doivent prendre des photographies des obstacles qu'ils rencontrent et les envoyer à un serveur central. En même temps que les photographies, les robots transmettent les informations qu'ils ont sur leurs positions. Ces informations peuvent être utilisées par les coordinateurs de l'opération de secours afin d'optimiser (*i.e.* accélérer et sécuriser) les interventions des secouristes humains, dotés de PDAs.

Nous adoptons naturellement une approche basée sur les Systèmes Multi-Agents (SMAs). Chaque robot est vu comme un agent. De cette manière, la

10. Autonomous Robots for Observation of Urban Networks after Disaster

11. Un odomètre est un capteur permettant de connaître la distance parcourue par un véhicule, sur la base du nombre de tours effectués par les roues.

flotte de robots peut être vue comme un SMA. Les agents/robots doivent mettre en œuvre deux tâches complémentaires :

- explorer la zone et collecter des observations à envoyer au serveur central ;
- acheminer les observations vers le serveur central à travers un réseau *ad hoc* mobile composé de robots.

6.4.1.3 Modélisation et hypothèses de travail

Nous modélisons deux sortes de robots. Les *robots d'exploration* se focalisent sur la collecte des observations (positions des obstacles) et sont chargés de transmettre ces données à des robots de communication. Les *robots de communication* se focalisent sur l'acheminement des observations vers un serveur central : ils jouent le rôle d'infrastructure de communication mobile.

Sachant que les interfaces WiFi des robots ont une portée limitée, il est clair que la flotte de robots ne va pas constituer naturellement un réseau connexe. Dans cette étude, nous supposons que (H1) *les robots de communication sont capables de se déployer en formant un groupe connexe* pour simplifier le problème et que (H2) *les robots d'exploration communiquent principalement sur des distances courtes* pour qu'ils consacrent principalement leur énergie aux nombreux déplacements qu'ils font.

6.4.2 Configurations

Avec MaDcAr-Agent, nous pouvons fournir différentes alternatives de comportement pour les robots. Par exemple, notre robot d'exploration comporte trois configurations : `Config1`, `Config2` et `Config3`. La figure 6.19 montre deux configurations pour l'exploration de la zone. `Config1` permet au robot d'explorer de manière aléatoire la zone de la mission et de transmettre ses « observations » (positions dégagées et positions des obstacles rencontrés) en direction d'un robot capable de l'acheminer vers le serveur central (*i.e.* utilisant `Config3`). `Config2` permet au robot de réaliser une exploration plus structurée (en l'occurrence un parcours en spirale) et de stocker ses observations. Par ailleurs, la figure 6.20 montre la configuration `Config3` qui est utilisée à la fois pour les robots d'exploration et les robots de communication [12]. L'utilité de `Config3` pour les robots de communication vient du fait que le rôle `DeployementExploration` doit concrétiser l'hypothèse $H1$ du paragraphe précédent. De plus, `Config3` est parfois utile pour des robots d'exploration qui se sont trop éloignés des robots de communication, puisque

12. Les robots de communication qui sont décrit uniquement avec `Config3` ne sont pas vraiment considérés comme étant auto-adaptables.

la portée de communication est maximale en l'adoptant (d'après l'hypothèse $H2$).

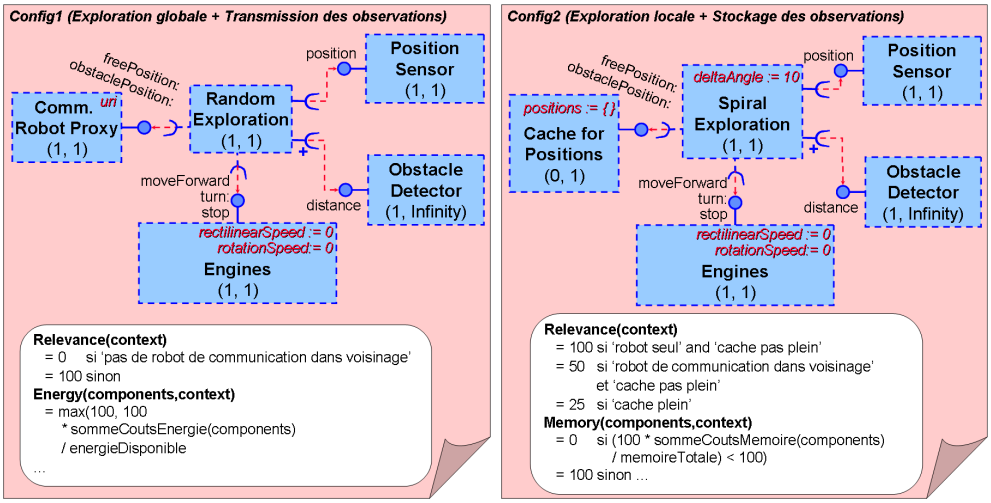

FIGURE 6.19 – Deux des configurations pour un robot d'exploration.

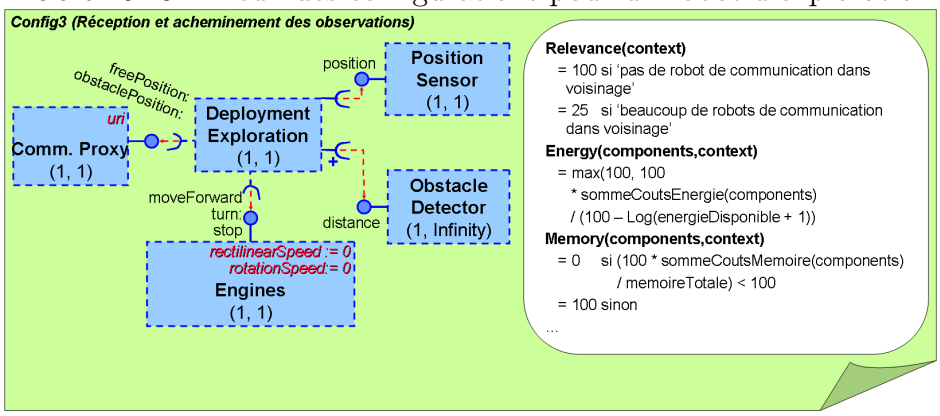

FIGURE 6.20 – Configuration utilisée pour les deux types de robots.

Les figures 6.20 et 6.20 donnent les fonctions de caractérisation pour les propriétés des trois configurations. Par exemple, la pertinence de ces configurations dépend du voisinage du robot (*i.e.* les robots assez proches pour une communication en WiFi) et aussi de certains paramètres internes tel que la quantité d'informations stockées en attendant d'être transmises. En particulier, la pertinence de la configuration `Config1` vaut 0 lorsqu'il n'y a pas de robot de communication dans le voisinage : il est inutile de tenter de transmettre les observations.

6.4.3 Contexte

```
neighborhoodSensor := Sensor new.
neighborhoodSensor resource: 'external/hardware/network/wifi/neighbors'.
neighborhoodSensor operation: #getNumberOfNeighbors.
neighborhoodSensor operation: #getNumberOfAppearedNeighbors.
. . .
energySensor := Sensor new.
energySensor resource: 'external/hardware/system/battery'.
energySensor operation: #getAmountOfAvailableEnergy updatePeriod: 1000.
energySensor operation: #getAmountOfMaxEnergy.
```

FIGURE 6.21 – Code du contexte du robot d'exploration.

La figure 6.21 fournit le code du contexte pour le robot d'exploration. Son utilisation par la politique d'assemblage est explicitée dans le paragraphe suivant.

6.4.4 Politique d'assemblage

La politique d'adaptation d'un agent dans MADCAR-AGENT se traduit par la spécification d'une politique d'assemblage et d'une politique de gestion du contenu. Un exemple de politique d'assemblage pour un robot d'exploration est donné dans la figure 6.22. Il se décompose de la manière suivante :

- `minDurationBetweenAdaptationTriggering` : le temps minimal entre deux réassemblages commence à 2 secondes et il augmente en fonction de la quantité d'énergie encore disponible dans les batteries du robot.
- `contextualSituations` : les conditions de déclenchement des réassemblages sont basées sur l'évolution du nombre de robots faisant parti du voisinage du robot à adapter et sur l'état des batteries du robot. La première situation contextuelle spécifiée représente le cas où la quantité d'énergie disponible est plutôt faible (inférieure à 20%) et des voisins sont présents (au moins un). Elle offre à l'agent l'opportunité de choisir une configuration plus économe et/ou supprimer des composants de l'assemblage. La deuxième situation contextuelle spécifiée représente le cas où le robot passe d'un voisinage minimal (un seul voisin) à un voisinage un peu plus conséquent (au moins deux). Elle offre notamment l'occasion de passer la troisième à la première configuration, car plus il y a de robots capables d'acheminer des observations dans le voisinage, plus on peut se permettre d'avoir des robots qui explorent la zone.
- `maxDurationForAdaptationDecision` : le temps maximal dédié à la décision des réassemblages vaut au moins 1 seconde et augmente de

```
"---------- Déclenchement ----------"
minDurationBetweenAdaptationTriggering := [ :contex |
  | availableEnergy maxEnergy |
  availableEnergy := (context at: #energySensor) getAmountOfAvailableEnergy.
  maxEnergy := (context at: #energySensor) getAmountOfMaxEnergy.
  2000 + 1000 * ((maxEnergy - availableEnergy) / maxEnergy) ].
contextualSituations := {
  (energySensor getAmountOfAvailableEnergy <= 20)
  & (neighborhoodSensor getNumberOfNeighbors >= 1).
  (neighborhoodSensor getNumberOfAppearedNeighbors >= 2)
  & (neighborhoodSensor getNumberOfPreviousNeighbors = 1).
  . . .
  }.
"----------- Décision -----------"
maxDurationForAdaptationDecision := [ :context |
  | availableCPU maxCPU |
  availableCPU := (context at: #cpuSensor) getAmountOfAvailableCPU.
  maxCPU := (context at: #cpuSensor) getAmountOfMaxCPU.
  1000 + 1000 * ((maxCPU - availableCPU) / maxCPU) ].
configAndComponentSelectingFunction := [ :config :components :context |
    (50 * config cfRelevance value: context)
  - (30 * config cfEnergy value: components value: context)
  - (10 * config cfCPU value: components value: context)
  - (10 * config cfMemory value: components value: context)
  ].
```

FIGURE 6.22 – Code de la politique d'assemblage d'un robot d'exploration.

plus en plus lorsque la quantité de CPU disponible diminue. C'est une conséquence du fait que le niveau méta a une priorité d'exécution inférieure au niveau de base, afin de perturber le moins possible le comportement applicatif.

– `configAndComponentSelectingFunction` : la fonction de sélection de la nouvelle configuration et des composants à utiliser met avant tout l'accent sur la pertinence de la configuration par rapport au contexte (coefficient de 50%). Elle tient aussi compte des besoins en énergie du nouvel assemblage (coefficient de 30%) et dans une moindre mesure, de ses besoins en CPU et en mémoire (coefficient de 10% chacun).

6.4.5 Politique de gestion de contenu

```
"---------- Ajout ou suppression de composants ----------"
minDurationBetweenContentChange := 5000.
maxDurationForNeighborhoodUpdate := 2000.
minMemoryForAllowingComponentAddition := 80.
componentUtilityFunction := [ :component :context |
  | pastUseRate futureUsefulness |
  pastUseRate      := [ :component :context | . . . ].
  futureUsefulness := [ :component :context | . . . ].
  (pastUseRate value: component value: context)
  / (futureUsefulness value: component value: context)
  ].
minUtilityForDenyingComponentDeletion := 20.
"-----------       Échange de composants       -----------"
maxDurationForWaitingAgentResponse := 500.
agentSociabilityFunction := [ :contex |.
  | availableCPU maxCPU |
  availableCPU := (context at: #cpuSensor) getAmountOfAvailableCPU.
  maxCPU := (context at: #cpuSensor) getAmountOfMaxCPU.
  (availableCPU / maxCPU) < 90 ].
rolePriorityFunction := [ :role :context |
  . . .
  ].
maxSizeOfPrioritaryRolesSet := 5.
```

FIGURE 6.23 – Code de la politique de gestion de contenu d'un robot.

Nous n'avons pas fait d'expérimentations concernant la gestion du contenu des agents car l'implémentation de cet aspect est encore en cours. Il est clair qu'en permettant aux robots d'échanger des composants, la gestion de la flotte de robots par le centre de commandement devient plus flexible. L'une des applications possibles de la gestion de contenu dans ce scénario est de

minimiser le nombre de composants fournis aux robots au début de la mission afin de pouvoir diffuser des composants plus spécifiques après les premières observations de robots. La figure 6.23 donne un échantillon du code pour la politique de gestion du contenu des robots d'exploration. En particulier, nous spécifions que :

- l'ajout de composant est autorisé lorsque la quantité de mémoire libre est au-dessus de 80% de la quantité maximale de mémoire (voir `minMemoryForAllowingComponentAddition`) ;
- l'agent autorise les échanges de composants (c'est-à-dire `agentSociabilityFunction` renvoie « vrai ») uniquement lorsque le taux d'utilisation de son CPU est inférieur à 90%.

6.4.6 Discussion

Pour réaliser le scénario décrit au paragraphe 6.4.1.2, un système multi-agent a été implémenté selon un approche « *Bottom-Up* » et centré-agent. En effet, chaque agent a été conçu de manière individuelle en spécifiant des configurations, des composants, une politique d'adaptation, etc. La politique d'adaptation de chaque agent leur permet de se spécialiser en fonction des changements contextuels. L'exécution d'un ensemble d'agents permet de faire émerger une organisation.

Nous avons simulé sur un PC de bureau [13] un terrain constitué d'obstacles et abritant des robots de communication (qui ont un comportement fixe) et des robots d'exploration (qui sont auto-adaptables). La figure 6.24 représente une simulation comportant ces deux types de robots : les robots de communication sont représentés avec une « aire de réception des observations » centrée sur eux et les robots d'exploration sont représentés avec des capteurs à l'avant. Cette aire de réception des observations est induite par la faible portée des communications des robots d'exploration utilisant les configurations `Config1` et `Config2`. Mais, les robots (de communication notamment) utilisant la configuration `Config3` ont entre eux une portée de communication plus importante. Dans la simulation, lorsqu'un robot d'exploration adopte la configuration `Config3` alors une aire de réception des observations est affichée autour de lui.

Lors de nos expérimentations, nous avons essayé d'évaluer la performance des réassemblages d'un agent comportant deux configurations (soit une dizaine de rôles), une douzaine de composants et une politique d'assemblage peu élaborée. Les mesures ont été effectuées sur un portable HP nx7000, Intel

13. Au moment de nos expérimentations, l'utilisation des wifibots n'était pas encore envisageable. Notamment, la machine virtuelle de Squeak n'avait pas encore été portée sur l'infrastructure des wifibots.

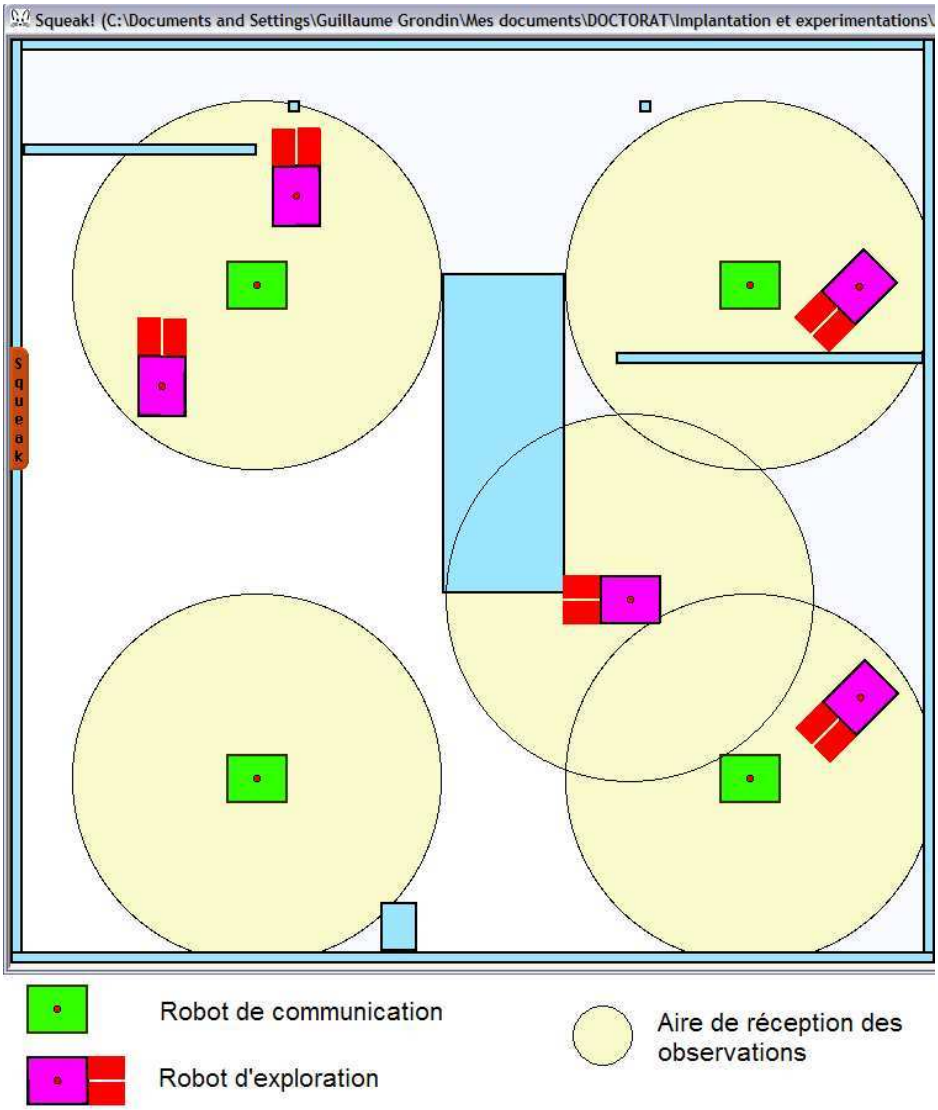

FIGURE 6.24 – Simulation impliquant plusieurs robots.

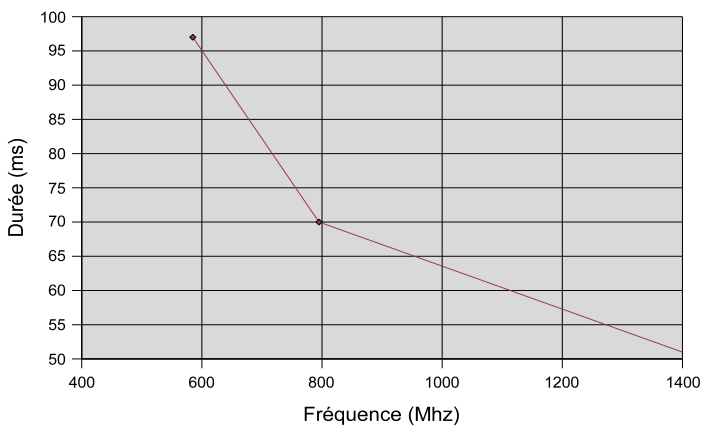

FIGURE 6.25 – Durée d'assemblage en fonction de la fréquence du processeur.

Centrino 1.4 GHz, doté 768 Mo de mémoire RAM et exécutant Squeak 10.2. Sur 1000 essais, le temps moyen d'un réassemblage était de 48 *ms* sachant que la configuration et les composants devaient changer systématiquement, et les valeurs mesurées allaient de 29 *ms* à 182 *ms*. Nous avons aussi analyser l'incidence de la fréquence du processeur sur la durée moyenne des réassemblages. La figure 6.25 montre que si le processeur est deux fois moins puissant, alors la durée des réassemblages ne double pas. Les valeurs mesurées sont acceptables car nous estimons que le contexte ne change pas souvent à une seconde d'intervalle. Cependant, nous sommes conscients qu'il faudra refaire des tests sur un scénario impliquant plusieurs robots communicants qui réalisent des tâches complexes (par exemple, des enregistrements vidéo).

6.5 Conclusion

Le modèle MADCAR est conçu pour faciliter non seulement l'adaptation d'une application par réassemblage de ses composants, mais aussi l'*évolution de l'application*, notamment en ajoutant ou en supprimant des configurations. La spécification des configurations est l'une des principales tâches d'un concepteur d'applications. Naturellement, plus une configuration est complexe (nombreux rôles, nombreux contrats pour décrire les rôles, etc.) plus son temps de conception est long. Cependant, une fois conçues, on peut les utiliser facilement dans différentes applications.

Nous avons mené des expérimentations pour valider notre approche en plusieurs phases. Une première phase d'expérimentation (section 6.2) a permis de voir certaines limites et d'améliorer et finaliser le modèle MADCAR.

C'est en grande partie pour simplifier la *réutilisation des configurations* que nous avons fait évoluer le modèle MaDcAr sur deux points.

L'une des évolutions du modèle MaDcAr concerne le transfert d'état (section 6.3). Nous avons utilisé un réseau de transfert d'état plutôt qu'une configuration pivot pour avoir un meilleur *découplage des configurations par rapport à l'aspect « description d'assemblages » et l'aspect « transfert d'état »*.

L'autre évolution importante du modèle MaDcAr concerne la prise en compte de différentes *propriétés extra-fonctionnelles* (CPU, mémoire, etc.) dans les décisions d'adaptation (section 6.3). Pour ce faire, les configurations sont dotées d'une fonction de caractérisation pour chacune des propriétés extra-fonctionnelles et pour la pertinence fonctionnelle par rapport à un contexte. Comme pour le réseau de transfert d'état, ces préoccupations supplémentaires allongent la tâche de spécification du concepteur de l'application, mais elles sont réutilisables dans différentes applications.
Par ailleurs, cette évolution a guidé la manière de spécifier la politique d'assemblage de l'application, en particulier à travers la notion de fonction de sélection. En effet, la politique d'assemblage est dotée d'un plus haut degré d'abstraction comparé à l'exemple de l'horloge adaptable (section 6.2). Il n'est plus nécessaire de faire des références directes à des configurations ou des rôles spécifiques pour décrire les adaptations. Donc, on peut très simplement ajouter et supprimer des configurations sans avoir à mettre à jour la politique d'adaptation.

Le modèle MaDcAr-Agent est conçu pour faciliter les adaptations des agents. En particulier, il permet facilement d'intégrer un comportement particulier dans les différents agents. Cette propriété a été mise à profit pour réaliser le scénario de la section 6.4, dans lequel il est utile que les deux types d'agents aient parfois le même comportement. Enfin, ce scénario a montré que le concepteur pouvait spécifier un comportement d'adaptation assez complexe à travers une politique d'adaptation concise et ayant un haut niveau d'abstraction.

Quatrième partie

Conclusion

Chapitre 7

Bilan et Perspectives

Bilan

Dans le cadre de cette thèse, nous nous sommes intéressés à la problématique de l'adaptation dynamique et automatique d'applications à base de composants en général et d'agents à base de composants en particulier. Pour ce faire nous avons étudié dans l'état de l'art différentes facettes de cette problématique en regardant tout d'abord l'adaptation dans les applications à base de composants puis l'adaptation dans les architectures d'agents. Nous avons ainsi mis en avant différents constats.

Critères de qualité de l'adaptation

Nous avons constaté dans notre étude que les adaptations dynamiques sont difficiles à mettre en œuvre, surtout lorsqu'il faut contrôler la qualité de ces adaptations, c'est-à-dire prendre en considération des **préoccupations extra-fonctionnelles** en supplément des besoins fonctionnels de l'application. Les principaux critères de qualité sont :

- le maintien de *cohérence*,
- la réduction des coûts des adaptations et donc la maximisation des *performances*,
- la maximisation de la *disponibilité* de certaines fonctionnalités d'une application lors d'une adaptation,
- la gestion de la *simultanéité* des adaptations en coordonnant automatiquement les adaptations qui se chevauchent sur le plan temporel ou le plan spatial, et enfin
- le critère d'*ouverture* qui symbolise la possibilité de paramétrer l'infrastructure afin d'établir un compromis entre les critères précédents.

Ainsi, une infrastructure d'adaptation doit idéalement utiliser des concepts de haut niveau comme des contrats, des configurations et des politiques d'adaptation.

Concilier autonomie et changements imprévus

Nous avons également constaté que l'adaptabilité d'un agent est particulièrement critique dans le cas où des changements contextuels peuvent être fréquents ou imprévisibles, comme par exemple dans un SMA ouvert. Plus précisément, un concepteur d'agents auto-adaptables doit savoir concilier (i) la *prise en charge des perturbations d'un système* dues à des changements imprévus et répétés avec (ii) le besoin d'*autonomie des agents*.

Les modèles d'agents auto-adaptables qui ne sont pas basés sur des composants logiciels ont généralement une architecture figée, d'où une difficulté pour faire évoluer sensiblement les agents déjà déployés dans un environnement ouvert. Au contraire, les modèles d'agents auto-adaptables à base de composants ont une **structure évolutive** qui simplifie notamment la reconfiguration d'un agent (par ajout, suppression et remplacement de composants) pour qu'il convienne le plus possible à son environnement changeant. Cependant, aucun des modèles étudiés ne vise explicitement à pouvoir passer automatiquement d'une architecture d'agent à une autre en fonction des contraintes extra-fonctionnelles de l'environnement (notamment, les variations de ressources matérielles).

Démarche suivie

La piste que nous avons identifiée pour résoudre ces problèmes est l'utilisation d'un modèle d'agents auto-adaptables à base de composants et muni d'une infrastructure d'adaptation qui prenne en compte les différents critères de qualité évoqués. Nous avons procédé en deux temps. Dans un premier temps, nous avons proposé un modèle pour la conception et l'adaptation d'applications à base de composants : le modèle MadcAr. Dans un second temps, nous avons proposé un modèle pour la conception d'agents auto-adaptables : le modèle MadcAr-Agent. Nous nous sommes efforcés de simplifier la spécification du « comportement d'adaptation » en favorisant des propriétés comme l'*abstraction* et la *réutilisabilité*.

MaDcAr : conception et adaptation d'applications à base de composants

Nous avons proposé MaDcAr[1], un modèle de **moteurs d'assemblage** dédiés à l'adaptation dynamique et automatique d'applications à base de composants. Ce modèle fournit une *solution uniforme*, basée sur un solveur de contraintes, permettant non seulement la construction d'applications par assemblage automatique mais aussi la reconfiguration dynamique de ces applications. La spécification des adaptations est à la fois globale à l'application et découplée des composants. De ce fait, l'approche MaDcAr présente l'avantage d'éviter les problèmes de cohérence des adaptations rencontrés dans les approches où chaque composant s'adapte indépendamment des autres [DL03, PBJ98]. De plus, les spécifications d'assemblage et d'adaptation peuvent être *réutilisées avec des jeux de composants différents*, et ce pour des composants fournissant des contrats sur les ressources matérielles (CPU, mémoire, énergie) qu'ils requièrent. Enfin, comme notre approche d'adaptation est *indépendante d'un modèle de composants particulier*, nous proposons un formalisme qui permet au concepteur de l'application de définir l'état de l'application et de spécifier les règles de *transfert d'état de manière générique*. Hormis l'existence de certaines interfaces extra-fonctionnelles (contrats sur les ressources matérielles, gestion de l'état et de l'activité d'un composant), aucune hypothèse n'est faite sur les composants ou sur le modèle de composants utilisé.

MaDcAr-Agent : utilisation des interactions entre agents pour optimiser leur auto-adaptabilité

Nous avons proposé MaDcAr-Agent, un modèle d'**agents auto-adaptables à base de composants** et muni d'une infrastructure dédiée à l'adaptation. La conception d'un agent s'organise autour de trois niveaux : le *niveau infrastructure* qui est l'interface entre l'agent et son infrastructure de déploiement, le *niveau de base* qui représente le comportement applicatif de l'agent sous la forme d'un assemblage de composants et le *niveau méta* qui comporte non seulement un moteur d'assemblage en charge des adaptations dynamiques et automatiques du niveau de base mais également un gestionnaire de contenu (*i.e.* composants). Le fonctionnement du niveau méta est guidé par la spécification de deux politiques : la *politique d'assemblage* qui permet à l'agent de s'adapter aux changements de contexte

1. « Model for Automatic and Dynamic Component Assembly Reconfiguration ».

en fonction des composants disponibles et la *politique de gestion de contenu* qui permet à l'agent d'obtenir les composants dont il a le plus besoin grâce aux interactions avec les autres agents. A travers ces spécifications explicites et découplées du comportement applicatif de l'agent, le concepteur d'agents décrit comment le système va prendre en charge les perturbations dues à des changements imprévus et répétés, sans pour autant nuire à l'autonomie des agents qui composent ce système.

Le fait d'utiliser des interactions entre agents pour la gestion de contenu des agents permet de surpasser l'adaptabilité d'une application isolée. Dans MaDcAr-Agent, les agents ont la possibilité de s'échanger des composants, ce qui leur offre une plus grande liberté pour gérer leur contenu car ils peuvent obtenir les composants dont ils ont besoin sans intervention humaine et ils peuvent supprimer des composants devenus trop encombrants ou inadaptés.

Perspectives

Il y a plusieurs aspects qui pourraient être approfondis dans notre approche :

- *simplifier la description d'une application :* la tâche du concepteur peut être facilitée en introduisant des mécanismes supplémentaires pour étendre des configurations (notamment en fragmentant une configuration en plusieurs morceaux réutilisables) et pour générer le réseau de transfert d'état (notamment en se basant sur un langage de contrats riches ou sur une ontologie) ;
- *optimiser les réassemblages qui n'impliquent pas de changement de configuration :* par exemple, en se restreignant aux opérations d'ajouts et de suppressions de composants selon les multiplicités des rôles ;
- *optimiser les réassemblages entre deux configurations proches :* par exemple, en restreignant l'adaptation aux différences entre ces deux configurations ;
- *étendre* MaDcAr *pour pouvoir adapter des composants composites :* par exemple, en modifiant le processus d'assemblage de façon à supprimer d'un composite les sous-composants superflus, étant donnés les besoins spécifiés dans le rôle à satisfaire ;
- *étendre* MaDcAr-Agent *pour pouvoir échanger des configurations entre agents :* si deux agents ont au moins une configuration en commun, alors il est probablement possible de mettre à jour automatiquement le réseau de transfert d'état de l'agent qui reçoit la nouvelle configuration.

Par ailleurs, nous pouvons envisager des perspectives à plus long terme concernant MaDcAr et MaDcAr-Agent.

Optimisations de MaDcAr pour l'informatique embarquée

Le modèle MaDcAr permet de concevoir des applications auto-adaptables dont les adaptations sont dirigées par une politique d'assemblage en fonction des changements de contexte. En particulier, le processus d'adaptation prend en compte le niveau des ressources matérielles disponibles pour déterminer dynamiquement les adaptations à réaliser.

Dans le cadre de l'informatique embarquée [Crn04, WDN05], il est souhaitable d'optimiser notre approche pour qu'elle nécessite le moins de ressources matérielles possible. Une piste envisageable est d'utiliser le moteur d'assemblage en mode statique (*i.e.* hors-ligne) pour réaliser l'essentiel des calculs avant le déploiement de l'application. Il s'agit de tester la compatibilité des rôles contenus dans les configurations avec les éléments d'une base globale de composants afin de :

- déterminer le meilleur ensemble de composants à intégrer dans l'application, c'est-à-dire le plus petit sous-ensemble de composants pouvant remplir les configurations, et
- pré-calculer l'ensemble des composants capables de jouer un rôle donné appartenant aux configurations de l'application.

Ces calculs hors-lignes permettent d'optimiser les décisions et la réalisation des adaptations, en minimisant l'ensemble de composants à utiliser en fonction des configurations présentes, et en calculant à l'avance toutes les adaptations possibles pour cette application.

Extension de MaDcAr-Agent pour définir des organisations

Actuellement, MaDcAr-Agent est caractérisé par une approche de conception centré-agent. Ainsi, la conception d'un SMA se fait en spécifiant, pour chaque agent, des configurations, des composants, une politique d'adaptation, etc. Néanmoins, cette approche ne permet pas de définir une organisation spécifique pour un SMA. L'aspect organisationnel d'un SMA offre plusieurs avantages dans le cadre d'une flotte de robots qui doivent réaliser des missions de secours (voir le Projet AROUND présenté au chapitre 6), notamment :

- la répartition des tâches parmi la flotte de robots, et
- la distribution efficace des composants nécessaires à ces nouvelles tâches.

Une piste envisageable est d'introduire le concept d'organisation dans notre approche. Il existe plusieurs modèles organisationnels pour SMA. Une partie considérable de ces modèles s'appuie sur la notion de « rôle d'agent » [2], par exemple le modèle AGR (Agent-Groupe-Rôle) [FG98]. L'utilisation d'une organisation dans MaDcAr-Agent permettra de paramétrer le niveau méta des agents selon les rôles-agents disponibles. Pour ce faire, chaque agent devra tenir compte des rôles-agents qu'il doit endosser (cas d'une organisation imposée) ou qu'il veut endosser (cas d'une organisation émergente) pour :

- influencer la politique d'assemblage, notamment le choix d'une configuration lors d'un réassemblage ;
- influencer la politique de gestion du contenu, notamment le choix des composants à obtenir ou à supprimer de manière prioritaire.

Notons que l'adaptation d'un agent appartenant à une organisation, de meme que l'ajout et la suppression d'agents constitue une opération de *réorganisation*. Une adaptation d'agent peut être requise après l'ajout ou la suppression dynamique d'autres agents dans une organisation. C'est pourquoi, il est primordial de disposer de mécanismes d'auto-adaptation chez chaque agent du système.

Pour conclure, ce travail souligne l'importance de l'auto-adaptation d'applications et d'agents pour les systèmes informatiques d'aujourd'hui et de demain. Gageons que ce problème de recherche sera déterminant à l'heure où les grands industriels [APC] investissent dans le futur de l'informatique ubiquiste et où la démocratisation des robots autonomes s'annonce [Gat07].

2. Pour éviter toute confusion avec les rôles qui décrivent des composants dans notre approche, un rôle d'agent est appelé rôle-agent.

Bibliographie

[AC87] P. Agre and D. Chapman. Pengi : An implementation of a theory of activity. In K. Forbus and H. Shrobe, editors, *AAAI'87 : Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 268–272. San Francisco, CA : Morgan Kaufmann, 1987. 58, 60, 61

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, 1997. 43

[Ajm04] Sameer Ajmani. *Automatic Software Upgrades for Distributed Systems*. PhD thesis, MIT, 2004. 37, 38, 42, 110

[AMM01] J. Arcangeli, C. Maurel, and F. Migeon. An api for high-level software engineering of distributed and mobil applications. In *FTDCS'01 : Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, page 155, Washington, DC, USA, 2001. IEEE Computer Society. 85

[AP] Projet AROUND. `http://www.ifi.auf.org/site/content/view/48/84/`. 155, 177

[APC] Ambience Project Consortium. `http://www.hitech-projects.com/euprojects/ambience/consortium_open.html`. 194

[BAS+04] A. Baumann, J. Appavoo, D. Silva, O. Krieger, and R. Wisniewski. Improving operating system availability with dynamic update. In *OASIS–Proceedings of the 1st Workshop on Operating System and Architectural Support for the On-demand IT InfraStructure, San Francisco, USA*, pages 21–27, October 2004. 46

[BBC02] Andrea Bracciali, Antonio Brogi, and Carlos Canal. Systematic component adaptation. *Electronic Notes in Theoretical Computer Science*, 66(4), 2002. 29

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self management in dynamic soft-

ware architecture specifications. In *WOSS'04 : Proceedings of the ACM SIGSOFT International Workshop on Self-Managed Systems*. ACM Press, October/November 2004. 34, 35

[BCP04] Antonio Brogi, Carlos Canal, and Ernesto Pimentel. Behavioural types and component adaptation. In *AMAST'04 : Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology*, pages 42–56, 2004. 29

[BCS02] E. Bruneton, T. Coupaye, and J. Stefani. Recursive and dynamic software composition with sharing. In *WCOP'02 : Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming*, Malaga, Spain, June 2002. 18, 156

[BDNP07] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, and Damien Pollet. *Squeak par l'Exemple*. Square Bracket Publishing, 2007. `http://www.iam.unibe.ch/~scg/SBE/fr/SBE.pdf`. 155

[Bel97] Luc Bellissard. *Construction et configuration d'applications réparties*. PhD thesis, Institut National Polytechnique de Grenoble, Decembre 1997. 41

[Ber01] L. Berger. *Mise en oeuvre des interactions en environnements distribués, compilés et fortement typés : le modèle MICADO*. PhD thesis, Université de Nice-Sophia Antipolis, Octobre 2001. 29

[BHA+05] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the 2005 USENIX Technical Conference*, Anaheim, CA, USA, April 2005. 109, 110

[BHP06] Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0 : Balancing advanced features in a hierarchical component model. In *SERA'06 : Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society. 31, 50, 51

[BIP88] M. E. Bratman, D. J. Israel, and M. E. Pollack. Plans and resource-bounded practical reasoning. *Computational Intelligence*, 4 :349–355, 1988. 60, 61

[BJPW99] Antoine Beugnard, Jean-Marc Jezequel, Noel Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7) :38–45, 1999. 103

[BJRV04] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture, European MDA Workshops : Foundations and Applications, MDAFA'03 and MDAFA'04*, pages 33–46, 2004. 99

[BJT02] Frances M. T. Brazier, Catholijn M. Jonker, and Jan Treur. Principles of component-based design of intelligent agents. *Data & Knowledge Engineering*, 41(1) :1–27, 2002. 63, 64

[BKB+04] L. Bölöni, M.A. Khan, X. Bai, G. Wang, Y. Ji, and D.C. Marinescu. Software engineering challenges for mutable agent systems. In *Software Engineering for Multi-Agent Systems II, Lecture Notes in Computer Science Vol 2940*, pages 149–167. Springer, 2004. 82

[BM00] L. Bölöni and D.C. Marinescu. Agent surgery : The case for mutable agents. In *Proceedings of the Third Workshop on Bio-Inspired Solutions to Parallel Processing Problems (BioSP3)*, May 2000. 82

[BM05] L. Bölöni and D.C. Marinescu. Adaptation and mutation in multi-agent systems and beyond. In *Design of Intelligent Multi-Agent Systems - Human Centeredness, Architectures, Learning and Adaptation*, pages 315–354. Springer, 2005. 6, 82

[BM06] Jean-Pierre Briot and Thomas Meurisse. A component-based model of agent behaviors for multi-agent-based simulations (short paper). In Luis Antunes and Keiki Takadama, editors, *5th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'06) 7th International Workshop on Multi-Agent-Based Simulation (MABS'06)*, pages 183–190, Hakodate, Japan, 5 2006. AAMAS. 79

[BMP06] Jean-Pierre Briot, Thomas Meurisse, and Frédéric Peschanski. Une expérience de conception et de composition de comportements d'agents à l'aide de composants. *L'OBJET*, 11(3), 2006. 6, 31, 58, 59, 62, 65, 79

[Boi01] O. Boissier. *Modèles et architectures d'agents*, chapter 2. Editions Hermès, dec 2001. 58, 59, 61, 62

[Bra03] Premysl Brada. *Specification-Based Component Substitutability and Revision Identification*. PhD thesis, Charles University in Prague, 2003. 37, 50

[Bro90] Rodney A. Brooks. A robust layered control system for a mobile robot. *Artificial intelligence at MIT : expanding frontiers*, pages 2–27, 1990. 58, 60

[BTV04] M.F. Bertoa, J.M. Troya, and A. Vallecillo. Usability metrics for software components. In *QAOOSE'04 : Proceedings of the 8th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, Oslo, Norway, 2004. 29

[CC07] Hervé Chang and Philippe Collet. Compositional Patterns of Non-Functional Properties for Contract Negotiation. *Journal of Software (JSW)*, 2(2) :12, August 2007. 99

[CCDG05] Joëlle Coutaz, James L. Crowley, Simon Dobson, and David Garlan. Context is key. *Commununication of the ACM*, 48(3) :49–53, 2005. 97, 101

[Crn01] Ivica Crnkovic. Component-based software engineering - new challenges in software development. *Software Focus*, December 2001. 13, 17, 18

[Crn04] Ivica Crnkovic. Component-based approach for embedded systems. In *WCOP'04 : In Proceedings of 9th International Workshop on Component-Oriented Programming*, 2004. 193

[CRS07] Denis Conan, Romain Rouvoy, and Lionel Seinturier. Scalable processing of context information with cosmos. In *DAIS'07 : Proceedings of the 7th IFIP International Conference on Distributed Applications and Interoperable Systems*, pages 210–224, 2007. 102

[Dav05] Pierre-Charles David. *Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation*. Phd thesis, Université de Nantes / École des Mines de Nantes, July 2005. 5, 47

[Dem01] Yves Demazeau. *VOYELLES*. Rapport d'Habilitation à Diriger des Recherches, Institut National Polytechnique de Grenoble, Laboratoire Leibniz, Avril 2001. 61, 65, 74

[DL03] Pierre-Charles David and Thomas Ledoux. Towards a framework for self-adaptive component-based applications. In *DAIS'03 : Proceedings of the 4th IFIP International Conference on Distributed Applications and Interoperable Systems*, volume 2893 of *LNCS*, pages 1–14. Springer-Verlag, 2003. 47, 96, 191

[DL05] Pierre-Charles David and Thomas Ledoux. Wildcat : a generic framework for context-aware applications. In *MPAC'05 :*

Proceeding of th 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing, Grenoble, France, November 2005. 47, 102, 172

[DSA01] A. Dey, D. Salber, and G. Abowd. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. *Special issue on context-aware computing in the Human-Computer Interaction Journal*, 16(2–4) :97–166, 2001. 25, 102

[EW92] M. Endler and J. Wei. Programming generic dynamic reconfigurations for distributed applications. In *Proceedings of the IE International Workshop on Configurable Distributed Systems*, pages 68–79. IEE, March 1992. 35

[Fab76] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *ICSE'76 : Proceedings of the 2nd international conference on Software engineering*, pages 470–476, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press. 40

[Fas01] Jean-Philippe Fassino. *THINK : vers une architecture de systèmes flexibles*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 2001. 48

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. 26

[Fer92a] I. A. Ferguson. *TouringMachines : An architecture for dynamic, rational, mobile agents*. PhD thesis, University of Cambridge, 1992. 6, 61, 67, 69, 71

[Fer92b] Innes A. Ferguson. Touring machines : Autonomous agents with attitudes. *Computer*, 25(5) :51–55, 1992. 67

[Fer95] Jacques Ferber. *Les systèmes multi-agents, vers une intelligence collective*. InterEditions, Paris (France), 1995. 57

[FG98] J. Ferber and O. Gutknecht. A meta-model for the analysis and design of organizations in multi-agent systems. In *ICMAS'98 : Proceedings of the 3rd International Conference on Multi Agent Systems*, page 128, Washington, DC, USA, 1998. IEEE Computer Society. 194

[FIP] Foundation for Intelligent Physical Agents. `http://www.fipa.org/`. 74

[FN71] Richard Fikes and Nils J. Nilsson. Strips : A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3/4) :189–208, 1971. 60

[FS91] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program : from concept to prototype. *Journal of Systems and Software*, 14(2) :111–128, 1991. 40

[Gat07] Bill Gates. A robot in every home. *Scientific American*, pages 58–65, January 2007. 194

[GF01] Olivier Gutknecht and Jacques Ferber. The madkit agent platform architecture. In *Revised Papers from the International Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, London, UK, 2001. Springer-Verlag. 74

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 27, 81

[GHM+99] Alexandre Guillemet, Grégory Haïk, Thomas Meurisse, Jean-Pierre Briot, and Marc Lhuillier. Mise en œuvre d'une approche componentielle pour la conception d'agents. In Marie-Pierre Gleizes and Pierre Marcenac, editors, *Septièmes Journées Francophones sur l'Intelligence Artificielle Distribuée et les Systèmes Multi-Agents (JFIADSMA'99)*. Hermès Science Publications, Paris, France, November 1999. 79

[GJ93] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software Practice and Experience*, 23(9) :949–964, 1993. 40

[GKdL04] Alessandro F. Garcia, Uirá Kulesza, and Carlos José Pereira de Lucena. Aspectizing multi-agent systems : From architecture to implementation. In *Software Engineering for Multi-Agent Systems III, Research Issues and Practical Applications [the book is a result of SELMAS 2004]*, pages 121–143, 2004. 63

[GL87] Michael P. Georgeff and Amy L. Lansky. Reactive reasoning and planning. In K. Forbus and H. Shrobe, editors, *AAAI'87 : Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 677–682. San Francisco, CA : Morgan Kaufmann, 1987. 60

[GRO02] Object Management GROUP. Corba components specification, version 3.0, omg tc document formal/2002-06-65, 2002. Available from `http://omg.org/cgi-bin/doc?formal/2002-06-65`. 18

[Gue96] Zahia Guessoum. *Un environnement opérationnel de conception et de réalisation de systèmes multi-agents*. Thèse de doctorat, Université Pierre et Marie Curie, France, 1996. 62

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3) :323–364, 1977. 87

[Hic01] Michael Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001. 40

[HT03] Petr Hnetynka and Petr Tuma. Fighting class name clashes in java component systems. In *JMLC*, pages 106–109, 2003. 26, 50

[IfSR] UCI Institute for Software Research. Architectural styles. `http://www.isr.uci.edu/architecture/styles.html`. 35

[IT02] P. Inverardi and M. Tivoli. Correct and automatic assembly of COTS components : an architectural approach. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering (CBSE5) : Benchmarks for Predictable Assembly*, 2002. 96

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2) :22–35, 1988. 156

[JW98] N. R. Jennings and M. Wooldridge. Applications of intelligent agents. *Agent technology : foundations, applications, and markets*, pages 3–28, 1998. 4, 57

[KBC02a] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre-Yves Cunin. Adapting applications on the fly. In *ASE'02 : Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, page 313, Washington, DC, USA, 2002. IEEE Computer Society. 95

[KBC02b] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre-Yves Cunin. Automatic adaptation of component-based software : Issues and experiences. In *PDPTA'02 : Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1365–1371. CSREA Press, 2002. 24

[KBY02] A. Ketfi, N. Belkhatir, and P. Y.Cunin. Adaptation dynamique, concepts et experimentations. In *ICSSEA'02 : Proceedings of the International Conference "Software and Systems Engineering and their Applications"*, Paris, France, 2002. 21

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1) :41–50, 2003. 26, 110

[Ket04] Abdelmadjid Ketfi. *Une approche générique pour la reconfiguration dynamique des applications à base de composants logiciels*. PhD thesis, Université Joseph Fourier, Décembre 2004. 36

[Kic96] Gregor Kiczales. Beyond the black box : Open implementation. *IEEE Software*, 13(1) :8–11, 1996. 39

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 : Proceedings of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997. 26, 175

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem : Dynamic change management. *IEEE Transaction on Software Engineering*, 16(11) :1293–1306, 1990. 34, 46, 109, 110

[KMW03] Richard Krutisch, Philipp Meier, and Martin Wirsing. The AgentComponent Approach, Combining Agents and Components. In Michael Schillo, Matthias Klusch, Jörg Müller, and Huaglory Tianfield, editors, *Multiagent System Technologies*, Lecture Notes in Computer Science 2831, pages 1–12. Springer-Verlag, Heidelberg, Germany, 2003. 63, 64

[KP04] Mejdi Kaddour and Laurent Pautet. Cooperative approach for mobile application adaptability based on mobilejms. In *UbiMob'05 : Proceedings of the 1st French-speaking conference on Mobility and ubiquity computing*, pages 174–181, 2004. 41

[Kum92] V. Kumar. Algorithms for constraint satisfaction problems : A survey. *AI Magazine*, 13(1) :32–44, 1992. 101

[LA06] Sébastien Leriche and Jean-Paul Arcangeli. Vers un modèle d'agent flexible. In *JMAC'06 : Journées Multi-Agent et Composant*, pages 49–59, Nîmes, France, Mars 2006. École des Mines d'Alès. 6, 85

[LA07] Sébastien Leriche and Jean-Paul Arcangeli. Adaptive Autonomous Agent Models for Open Distributed Systems. In *International Multi-Conference on Computing in the Global Information Technology (ICCGI), Guadeloupe, 04/03/2007-09/03/2007*, pages 19–24, http ://www.computer.org, mars 2007. IEEE Computer Society. 85

[LBB+01] T. Ledoux, M. Blay, E. Bruneton, D. Caromel, T. Coupaye, D. Hagimont, J-M. Menaud, J. Noyé, and M. Riveill. Etat de

l'art sur l'adaptabilité. Livrable D1.1, RNTL ARCAD, Ecole des Mines de Nantes, Décembre 2001. 21, 22

[Ler06] Sébastien Leriche. *Architectures à composants et agents pour la conception d'applications réparties adaptables*. Thèse de doctorat, Université Paul Sabatier, Toulouse, France, décembre 2006. 85

[LH05] O. Layaida and D. Hagimont. Composition et reconfiguration hiérarchiques pour des services multimédia auto-adaptables. In *Actes de la 4ème Conférence Française sur les Systèmes d'Exploitation (CFSE 4)*, Le Croisic, France, April 2005. 16

[Lhu98] Marc Lhuillier. *Une approche à base de composants logiciels pour la conception d'agents, Principes et mise en œuvre à travers la plate-forme MALEVA*. PhD thesis, Université de Paris 6, Février 1998. 79

[LP01] Irvin J. Lustig and Jean-François Puget. Program does not equal program : Constraint programming and its relationship to mathematical programming. *Interfaces*, 31(6) :29–53, 2001. 101, 108

[LPH04] Hua Liu, Manish Parashar, and Salim Hariri. A component-based programming model for autonomic applications. In *ICAC'04 : Proceedings of the First International Conference on Autonomic Computing*, pages 10–17, New York, USA, 2004. IEEE Computer Society. 31

[LRZJ04] Neil Loughran, Awais Rashid, Weishan Zhang, and Stan Jarzabek. Supporting product line evolution with framed aspects. In *ACP4IS'04 : Proceedings of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Lancaster, UK, March 2004. 27

[LV95] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, 1995. 43

[MB01] Thomas Meurisse and Jean-Pierre Briot. Une approche à base de composants pour la conception d'agents. *Technique et Science Informatiques (TSI)*, 20(4) :583–602, 2001. 79

[MDK94] Jeff Magee, Naranker Dulay, and Jeff Kramer. A Constructive Development Environment for Parallel and Distributed Programs. In *Proceedings 2nd IEEE International Workshop on Configurable Distributed Systems (IWCDS-2)*, 1994. 5, 43, 45

[Med96] Nenad Medvidovic. Adls and dynamic architecture changes. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints'96) on SIGSOFT'96 workshops*, pages 24–27, 1996. 34, 35, 43, 110

[Med99] Nenad Medvidovic. *Architecture-based specification-time software evolution*. PhD thesis, University of California, Irvine, 1999. Chair-Richard N. Taylor. 41

[Mei03] Philipp Meier. Visual construction of multi-agent-systems according to the agentcomponent approach and the run-design-time concept. In *Proceedings of the ACIS 4th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'03)*, pages 32–46, Lübeck, Germany, October 2003. International Association for Computer and Information Science. 64

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10) :40–51, 1992. 103

[MG03] Arun Mukhija and Martin Glinz. Casa - a contract-based adaptive software architecture framework. In *ASWN'03 : Proceedings of the 3rd IEEE Workshop on Applications and Services in Wireless Networks*, pages 275–286, Berne, Switzerland,, July 2003. IEEE Computer Society. 31, 52

[MG05] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *ARCS'05 : Proceedings of the 18th International Conference on Architecture of Computing Systems*, pages 124–138, Innsbruck, Austria, March 2005. 5, 52

[MMMS01] Annie Marcoux, Christine Maurel, Frédéric Migeon, and Patrick Sallé. Generic operational decomposition for concurrent systems : semantics and reflection. *Progress in computer research*, pages 225–242, 2001. 85

[MORT96] Nenad Medvidovic, Peyman Oreizy, Jason E. Robbins, and Richard N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *SIGSOFT'96 : Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*, pages 24–32, New York, NY, USA, 1996. ACM Press. 43

[MS89] Jeff Magee and Morris Sloman. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, 15(6) :663–675, 1989. 45

[MT97] Nenad Medvidovic and Richard N. Taylor. A framework for classifying and comparing architecture description languages. In *ESEC'97/FSE-5 : Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 60–76, New York, NY, USA, 1997. Springer-Verlag. 5, 40, 41, 43

[Mül96] J. P. Müller. *The Design of Intelligent Agents : A Layered Approach*. Number 1177 in Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, 1996. 6, 61, 69, 71

[Mül97] Jörg P. Müller. Control architectures for autonomous and interacting agents : A survey. In *PRICAI'96 : Proceedings from the Workshop on Intelligent Agent Systems, Theoretical and Practical Issues*, pages 1–26, London, UK, 1997. Springer-Verlag. 58, 59, 62

[NCCR05] Jayaprakash Nagapraveen, Thierry Coupaye, Christine Collet, and Nicolas Rivierre. Des règles actives au sein d'une infrastructure logicielle autonomique. In *RENPAR'16 / CFSE'4 / SympAAA'2005 / Journées Composants 2005*, Le Croisic, France, Avril 2005. 26

[NNL98] H.S. Nwana, D.T. Ndumu, and L.C. Lee. Zeus : An advanced tool-kit for engineering distributed multi-agent systems. In *PAAMS'98 : Proceedings of the Practical Application of Intelligent Agents and Multi-Agent Systems*, pages 377–392, 1998. 60

[OBDK02] Michel Occello, Christof Baeijs, Yves Demazeau, and Jean-Luc Koning. Mask : An aeio toolbox to develop multi-agent systems. In *Knowledge Engineering and Agent Technology, IOS Series on Frontiers in AI and Applications*, Amsterdam, The Netherlands, 2002. 63, 64

[Obj02] ObjectWeb. The fractal project, 2002. `http://fractal.objectweb.org/`. 18

[Obj06] ObjectWeb. Open ccm : The open corba component model platform, 2006. `http://openccm.objectweb.org/`. 18

[OFTA02] Observatoire Français des Techniques Avancées (OFTA). Les Systèmes Multi-Agents, 2002. `http://www.ofta.net/`. 57

[Ore96] Peyman Oreizy. Issues in the runtime modification of software architectures. Technical Report UCI-ICS-96-35, University of California, Irvine, Irvine, CA, USA, August 1996. 41

[OT98] P. Oreizy and R. Taylor. On the role of software architectures in runtime system reconfiguration. In *CDS'98 : Proceedings of the International Conference on Configurable Distributed Systems*, page 61, Washington, DC, USA, 1998. IEEE Computer Society. 43, 44

[PBJ98] F. Plasil, D. Balek, and R. Janecek. Sofa/dcup : Architecture for component trading and dynamic update. In *ICCDS'98 : Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems*, pages 35–42, May 1998. 5, 50, 51, 96, 191

[PGA02] Andrei Popovici, Thomas Gross, and Gustavo Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD'02 : Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press. 53

[RA00] Tobias Ritzau and Jesper Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, 2000. 14, 21, 41

[Ran02] Stephen Rank. *A Reflective Architecture to Support Dynamic Software Evolution*. PhD thesis, University of Durham, 2002. 26

[RD02] Pierre-Michel Ricordel and Yves Demazeau. La plate-forme volcano : modularité et réutilisabilité pour les systèmes multi-agents. *Numéro spécial sur les plates-formes de développement SMA. Revue Technique et Science Informatiques (TSI)*, 2002. 63, 64

[RG91] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a bdi-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *KR'91 : Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc. : San Mateo, CA, USA, 1991. 60

[RLP00] P. Roy, A. Liret, and F. Pachet. The framework approach for constraint satisfaction. *ACM Computing Surveys*, 32(1es), 2000. 101

[RMS01] Jean-Christophe Routier, Philippe Mathieu, and Yann Secq. Dynamic skill learning : A support to agent evolution. In *Procee-*

dings of the Artificial Intelligence and the Simulation of Behaviour symposium on Adaptive Agents and Multi-agent systems (AISB'01), pages 25–32, march 2001. 77

[RP97] Pierre Roy and François Pachet. Reifying constraint satisfaction in smalltalk. *JOOP*, 10(4) :43–51, 63, 1997. 156

[RW91] S. Russell and E. Wefald. *Do the right Thing*. MIT Press, 1991. 6, 71

[SCS02] A. Senart, O. Charra, and J.-B. Stefani. Developing dynamically reconfigurable operating system kernels with the think component architecture. In *ECOOSE' 02 : Proceedings of the Workshop on Engineering Context-aware Object-Oriented Systems and Environments, in association with OOPSLA'02*, Seattle, USA, November 2002. 48

[Sec03] Yann Secq. *RIO : Rôles, Interactions et Organisations, une méthodologie pour les systèmes multi-agents ouverts*. PhD thesis, Université des Sciences et Technologies de Lille, Lille, France, Décembre 2003. 6, 58, 77

[Sen03] Aline Senart. *Canevas logiciel pour la construction d'infrastructures logicielles dynamiquement adaptables*. PhD thesis, Institut National Polytechnique de Grenoble, November 2003. 5, 21, 23, 29, 30, 34, 48

[SF93] Mark E. Segal and Ophir Frieder. On-the-fly program modification : Systems for dynamic updating. *IEEE Software*, 10(2) :53–65, 1993. 40, 111

[SG96] Mary Shaw and David Garlan. *Software architecture : perspectives on an emerging discipline*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. 62

[SMBV03] I. Sora, F. Matthijs, Y. Berbers, and P. Verbaeten. Automatic composition of systems from components with anonymous dependencies specified by semantic-unaware properties. *Technology of Object-Oriented Languages, Systems & Architectures*, 732 :154–179, March 2003. 96

[Smi82] B. C. Smith. *Procedural reflection in programming languages*. PhD thesis, MIT Laboratory for Computer Science, Cambridge, MA, 1982. 26, 87

[Szy98] Clemens Szyperski. *Component software : beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998. 3, 15, 63

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 78, 83, 88, 95

[Uni05] United Nations. *Proceedings of World Conference on Disaster Reduction (WCDR'05)*, Kobe, Japan, January 2005. Proceedings available from `http://www.unisdr.org/wcdr/thematic-sessions/WCDR-proceedings-of-the-Conference.pdf`. 177

[VB03] Yves Vandewoude and Yolande Berbers. Meta model driven state transfer in component oriented systems. In *Proceedings of The Second International Workshop On Unanticipated Software Evolution*, pages 3–8, Warshau, Poland, April 2003. 113

[Ver04] L. Vercouter. MAST : Un modèle de composants pour la conception de SMA . In *Journées Multi-Agents et Composants (JMAC'04)*, pages 18–31, Paris, France, Novembre 2004. École des Mines de Paris. 6, 74

[VHT00] A. Vallecillo, J. Hernandez, and J. Troya. Component interoperability. Technical Report ITI-2000-37, Departmento de Lenguajes y Ciencias de la Computacion, University of Malaga, 2000. 29

[VS04] W. Vanderperren and D. Suvee. Jascoap : Adaptive programming for component-based software engineering. In *ACP4IS'04 : Proceedings of the 3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Lancaster, UK, March 2004. 27

[WB] Wifibot. `http://www.wifibot.com/`. 177

[WBLS01] M. Woodman, O. Benediktsson, B. Lefever, and F. Stallinger. Issues of cbd product quality and process quality. In *ICSE'01 : Proceedings of the 4th International Workshop of Component-Based Software Engineering at 23rd International Conference on Software Engineering*, Toronto, Canada, 2001. 29

[WDN05] Roel Wuyts, Stéphane Ducasse, and Oscar Nierstrasz. A data-centric approach to composing embedded, real-time software components. *J. Syst. Softw.*, 74(1) :25–34, 2005. 193

[Wei93] M. Weiser. Ubiquitous computing. *Computer*, 26(10) :71–72, 1993. 3

[WJ95] M. Wooldridge and Nicholas R. Jennings. Intelligent agents : Theory and practice. In *The Knowledge Engineering Review*, volume 2, pages 115–152. UMBC, 1995. 58, 59, 123

[WLF01] Michel Wermelinger, Antonia Lopes, and Jose Luiz Fiadeiro. A graph based architectural (re)configuration language. In *ESEC/FSE-9 : Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–32, 2001. 34, 35

[Zil96] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3) :73–83, 1996. 107

[ZYCM04] Ji Zhang, Zhenxiao Yang, Betty H.C. Cheng, and Philip K. McKinley. Adding safeness to dynamic adaptation techniques. In *Proceedings of ICSE'04 Workshop on Architecting Dependable Systems*, Edinburgh, Scotland, UK, May 2004. 27

École Nationale Supérieure des Mines
de Saint-Étienne

N° d'ordre : 499 I.

Guillaume Grondin

MaDcAr-Agent: a model of self-adaptive component-based agents

Speciality : Computer Science

Key words : Multi-Agent System, Software Components, Self-adaptation, Dynamic Adaptation, Open System, Assembling Engine

Abstract :

In the context of ubiquitous computing, the application execution environment is made of heterogeneous devices belonging to different users (PC, PDA, mobile phone, etc.). These characteristics require to structure the application in a network of software units that are relatively independent but may interact with each other. In this thesis, we propose MaDcAr-Agent, a model of self-adaptive component-based agents provided with a framework dedicated to adaptation. This model is characterized by a meta level which contains an assembling engine in charge of automatic and dynamic adaptations according to the agent's context. The behavior of the meta level is ruled by two specified policies: the *assembling policy* that allows an agent to adapt itself because of contextual changes according to the available components and the *content management policy* that allows an agent to obtain the components it mostly needs, by using interactions with other agents. These specifications are made explicit and uncoupled from the agent's applicative behavior. The agent designer uses them to take into account the disturbance of the system from unpredictible and repeated changes, without having a detrimental effect on the agent's autonomy. To validate our approach, several experimentations have been conducted with this model. For instance, we describe a scenario involving mobile robots that must explore autonomously an unknown area.

École Nationale Supérieure des Mines
de Saint-Étienne

N° d'ordre : 499 I.

Guillaume Grondin

MaDcAr-Agent : un modèle d'agents auto-adaptables à base de composants

Spécialité : Informatique

Mots clefs : Système Multi-Agent, Composants logiciels, Auto-adaptation, Adaptation dynamique, Système ouvert, Moteur d'assemblage

Résumé :

Dans le cadre de l'informatique ubiquiste, l'environnement d'exécution d'une application est constitué de machines hétérogènes en ressources matérielles et appartenant à des utilisateurs différents (PC, PDA, téléphone mobile, etc.). Ces caractéristiques imposent de structurer l'application en une organisation d'unités logicielles relativement indépendantes qui coopèrent et interagissent. Dans cette thèse, nous proposons MaDcAr-Agent, un modèle d'agents auto-adaptables à base de composants et muni d'une infrastructure dédiée à l'adaptation. Ce modèle se caractérise par la présence d'un niveau méta qui comporte notamment un moteur d'assemblage en charge des adaptations dynamiques et automatiques en fonction du contexte de l'agent. Le fonctionnement du niveau méta est guidé par la spécification de deux politiques : la *politique d'assemblage* qui permet à l'agent de s'adapter aux changements de contexte en fonction des composants disponibles et la *politique de gestion de contenu* qui permet à l'agent d'avoir les composants dont il a le plus besoin grâce aux interactions avec les autres agents. A travers ces spécifications explicites et découplées du comportement applicatif de l'agent, le concepteur d'agents peut prendre en charge la perturbation d'un système dû à des changements imprévus et répétés, sans pour autant nuire à l'autonomie des agents qui composent ce système. Pour valider notre approche, diverses expérimentations ont été menées avec ce modèle, notamment dans le cadre d'un scénario impliquant des robots mobiles qui doivent explorer une zone inconnue.